

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**DISCERN: A DISTRIBUTED ARTIFICIAL NEURAL NETWORK
MODEL OF SCRIPT PROCESSING AND MEMORY**

Risto Pekka Miikkulainen

**September 1990
CSD-900025**

UNIVERSITY OF CALIFORNIA

Los Angeles

**DISCERN: A Distributed Artificial Neural
Network Model
of Script Processing and Memory**

A dissertation

submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Risto Pekka Miikkulainen

1990

Copyright © 1990 by Risto Pekka Miikkulainen

All Rights Reserved

To my parents,
Raili and Pentti Miikkulainen

Table of Contents

I OVERVIEW	1
1 Introduction	2
1.1 Task	2
1.2 Motivation and goals	3
1.3 Approach	5
1.4 Guide to the reader	6
2 Background	8
2.1 Scripts	8
2.2 Parallel distributed processing	9
3 Overview of DISCERN	13
3.1 System architecture	13
3.1.1 Lexicon	13
3.1.2 Parsing subsystem	15
3.1.3 Episodic memory	16
3.1.4 Generating subsystem	16
3.1.5 Question answering subsystem	18
3.2 I/O example	18
3.3 Training and performance	20
II PROCESSING MECHANISMS	23
4 Developing representations in FGREP-modules	24
4.1 Backpropagation networks	24
4.2 FGREP: forming global representations with extended backpropagation . . .	27
4.2.1 Basic FGREP architecture	27
4.2.2 Extending backpropagation to the representations	28
4.2.3 Reactive training environment	28
4.3 Subtask: Assigning case roles to sentence constituents	29

4.4	Properties of FGREP-representations	32
4.4.1	Simulations	32
4.4.2	Final representations	33
4.4.3	Performance in the task	38
4.4.4	Damage resistance	40
4.4.5	Creating expectations about possible contexts	40
4.4.6	Generalization	41
4.5	ID+content: cloning synonymous word instances	44
4.5.1	Meaning and identity	44
4.5.2	Composing instances from ID and Content	45
4.5.3	Performance with cloned synonyms	46
4.5.4	Extending the vocabulary	47
4.6	Processing sequential input/output: The recurrent FGREP module	49
4.6.1	Representing constituent structures	50
4.6.2	Recurrent FGREP module	51
4.6.3	Case-role assignment from sequential input	52
4.6.4	Sequential expectations	53
4.7	Limitations	53
5	Building from FGREP modules	55
5.1	Performance phase	55
5.2	Training phase	59
5.3	Processing modules in DISCERN	61
5.3.1	Training data	61
5.3.2	Simulation set-up	62
5.3.3	Representations	63
5.3.4	Paraphrasing	63
5.3.5	Question answering	65
III	MEMORY MECHANISMS	69
6	Self-organizing feature maps	70
6.1	Topological feature maps	70
6.2	Self-organization	71
6.2.1	General mechanism	71

6.2.2	Biological model	73
6.2.3	Computational abstractions	74
6.2.4	Example process	75
6.3	Feature maps as memory models	77
7	Episodic memory organization: hierarchical feature maps	78
7.1	The hierarchical feature map architecture	78
7.2	Hierarchical feature maps in DISCERN	82
7.2.1	Organizing episodic memory	82
7.2.2	Representation of the role binding space	83
7.2.3	Retrieving representations	86
7.3	Properties	89
7.3.1	Memory organization	89
7.3.2	Self-organization	90
8	Episodic memory storage and retrieval: Trace feature maps	93
8.1	Trace feature maps	93
8.1.1	Initial set-up	93
8.1.2	Storage mechanism	94
8.1.3	Retrieval mechanism	95
8.1.4	Memory effects and capacity	96
8.1.5	Tuning the parameters	99
8.2	Trace feature maps in DISCERN	100
8.2.1	Storage	100
8.2.2	Retrieval	101
8.3	Storage and retrieval from episodic memory	102
8.3.1	Storing representations	102
8.3.2	Basic retrieval	104
8.3.3	Ambiguous retrieval	107
9	Lexicon	109
9.1	Overview of the architecture	109
9.2	Representation of lexical symbols	110
9.3	Properties of the lexicon model	110
9.3.1	Sentence training data	111

9.3.2	Lexical and semantic maps	112
9.3.3	Word associations	114
9.3.4	Transforming representations	115
9.3.5	Priming	116
9.3.6	Errors	117
9.4	The lexicon in DISCERN	118
9.4.1	Training the lexicon	118
9.4.2	Filtering out noise	121
9.5	Modeling the human lexical system	122
9.5.1	The DISLEX architecture	124
9.5.2	Modeling aphasia	125

IV EVALUATION 127

10	Behaviour of the complete model	128
10.1	Connecting the modules	128
10.2	Example run	133
10.2.1	Parsing	133
10.2.2	Paraphrasing	134
10.2.3	Basic question answering	135
10.2.4	Disambiguating the question	139
10.2.5	Ambiguous questions	141
10.2.6	Misleading and incorrect questions	144
10.3	Cleaning up errors	146
10.3.1	Tokens by story parser	146
10.3.2	Semantics by episodic memory	148
10.3.3	Semantics by FGREP modules	148
10.3.4	Input and output representations by lexicon	150
10.4	Error behaviour	151
10.4.1	Dealing with incomplete input	151
10.4.2	Role binding errors	153
10.4.3	Slips of tongue	155
10.4.4	Forgetting and memory confusions	158
11	Discussion	160

11.1	DISCERN as a physical model	160
11.2	Psychological implications	161
11.3	DISCERN as a developmental model	162
11.4	Making use of modularity	163
11.5	The role of the central lexicon	165
11.6	Robustness and stability	167
11.7	Question answering	167
11.8	Exceptions and novel situations	168
12	Comparison to previous work	170
12.1	Symbolic models of natural language processing	170
12.2	Parallel distributed models of NLP	172
12.2.1	Sentence processing models	172
12.2.2	Script processing models	173
12.2.3	Scripts in symbolic vs. PDP processing	174
12.2.4	Linguistic models	175
12.3	Localist models	176
12.4	Models of the lexicon	177
12.5	Models of episodic memory	178
12.6	Issues in artificial neural network modeling	180
12.6.1	Methods for forming distributed representations	180
12.6.2	Processing types and tokens	183
12.6.3	Sequential processing	184
12.6.4	Modularity	185
12.6.5	One-shot storage in associative memory	186
12.6.6	Role binding	187
12.6.7	Parallel distributed inference	189
13	Extensions and future work	191
13.1	Processing more complex texts	191
13.1.1	Pronoun reference	191
13.1.2	Multiple scripts	191
13.1.3	More complex syntax	192
13.1.4	Translation	194
13.2	Semantic representations	195

13.2.1	Cumulative representations	195
13.2.2	Composite representations	195
13.3	Lexicon	196
13.3.1	Extending the lexicon	196
13.3.2	Combining FGREP with the lexicon	196
13.3.3	Disambiguation	197
13.4	Episodic memory	198
13.4.1	Extensions to hierarchical feature maps	198
13.4.2	Extensions to the trace feature map mechanism	199
13.5	Question answering	200
13.6	Implementing control with networks	201
13.7	Discovering scripts	202
13.8	Representing and learning knowledge structures	203
14	Conclusions	204
14.1	Summary	204
14.2	Conclusion	207
A	Story data	221
B	Implementation details	226
C	DISCERN training code and data	227
C.1	FGREP modules	227
C.1.1	Training code	227
C.1.2	Generating training data	234
C.1.3	Training data	238
C.2	Hierarchical feature maps	241
C.2.1	Training code	241
C.2.2	Training data	249
C.3	Lexicon	250
C.3.1	Training code	250
C.3.2	Generating training data	256
C.3.3	Training data	258
D	DISCERN performance code and data	261

List of Figures

2.1	Localist and distributed representations	10
3.1	DISCERN block diagram (performance)	14
3.2	Lexicon	14
3.3	Sentence representation	15
3.4	Story representation	16
3.5	Episodic memory	17
3.6	Question representation	18
3.7	Cue representation	18
3.8	Answer representation	18
3.9	DISCERN block diagram (training)	20
4.1	Forward and backward propagation	25
4.2	Basic FGREP architecture	27
4.3	Case-role assignment	30
4.4	Basic FGREP snapshot	32
4.5	Representations, basic FGREP	33
4.6	Merge clustering	34
4.7	Single unit categorization	35
4.8	Two-unit categorization	36
4.9	Feature map of representations	37
4.10	Damage resistance	41
4.11	Expectations	42
4.12	Generalization with training set size	43
4.13	Cloning word instances	45
4.14	Representations, cloned synonyms	47
4.15	Performance, extended vocabulary	49
4.16	Recurrent FGREP module	51
4.17	Recurrent FGREP snapshot	52
5.1	Parsers	56

5.2	Generators	57
5.3	Cue former	58
5.4	Answer producer	59
5.5	Training FGREP modules	60
6.1	A self-organizing feature map network	71
6.2	Single level mapping	72
6.3	Self-organizing the weight vectors	76
7.1	Hierarchy of script based stories	79
7.2	Hierarchical feature map architecture	79
7.3	Compression of the input lines	81
7.4	Variance on the input lines	81
7.5	Hierarchical feature mapping of script-based stories	83
7.6	Role binding maps for the restaurant script	84
7.7	Role binding maps for the shopping script	85
7.8	Role binding maps for the travel script	85
7.9	Retrieving story representations	87
7.10	Representation accuracy with noisy input	89
7.11	Continuum of classification systems	91
8.1	Creating a trace	95
8.2	Retrieving a trace	96
8.3	Interaction of nearby traces	97
8.4	Overloading the memory	98
8.5	Creating trace for fancy-restaurant-JLMB	101
8.6	Retrieving fancy-restaurant-JLMB	102
8.7	Creating a trace for fancy-restaurant-MLLB	103
8.8	Competing traces	103
8.9	Unsuccessful retrieval	104
8.10	Storing RFJLMB	105
8.11	Retrieving RFJLMB	106
9.1	Lexical and semantic feature maps	110
9.2	Training data	111
9.3	Lexical map	113

D.1	The DISCERN program	261
D.1.1	Definitions	261
D.1.2	Main control	264
D.1.3	FGREP modules	267
D.1.4	Lexicon	271
D.1.5	Hierarchical feature maps	273
D.1.6	Trace feature maps	276
D.2	Data generation	278
D.3	Test data	280
D.3.1	Network files, representations and vocabulary	280
D.3.2	Complete stories	282
D.3.3	Incomplete stories	284
D.3.4	Example run	285
E	Analysis and comparison code and data	289
E.1	Case-role assignment with FGREP	289
E.1.1	Training code	289
E.1.2	Testing code	292
E.1.3	Generating data	298
E.1.4	Training data	301
E.1.5	Testing data	302
E.1.6	Sequential input	303
E.2	Merge clustering	304
E.2.1	Code	304
E.2.2	Data files	305
E.3	Single-level feature maps	305
E.3.1	Mapping of the FGREP representations	306
E.3.2	Single-level mapping of script-based stories	306
E.4	Sentence data for the lexicon	307
E.4.1	Generating training data	307
E.4.2	Data files	308
E.5	Ideal trace feature maps	309
E.5.1	Code	310
E.5.2	Data files	315

9.4	Semantic map	113
9.5	Lexical → semantic connections	116
9.6	Semantic → lexical connections	117
9.7	Lexical map of DISCERN	119
9.8	Semantic map of DISCERN	120
9.9	Correcting a noisy input representation	122
9.10	Number of correct words with noisy input	123
9.11	Representation accuracy with noisy input	123
9.12	Model of human lexical system	124
11.1	Hierarchical vs. single processing networks	164
12.1	Sentence gestalt model	173
12.2	Semantic feature encoding	181
12.3	Developing representations in hidden layers	182
12.4	Role bindings with tensorproducts	188
13.1	Block diagram of the relative clause processing system	192
13.2	Parsing relative clause structures	193
14.1	Main features of DISCERN	205

List of Tables

2.1	Causal chain and role bindings	9
4.1	Sentence generators	31
4.2	Noun categories	31
4.3	Performance, familiar	38
4.4	Performance, unfamiliar	39
4.5	Performance, basic FGREP	45
4.6	Performance, cloned synonyms	48
4.7	Recurrent FGREP performance	53
5.1	Paraphrasing performance in isolation	63
5.2	Paraphrasing performance in a chain	64
5.3	Paraphrasing performance incomplete stories	64
5.4	Q/A performance in isolation	66
5.5	Q/A performance in a chain	66
5.6	Q/A performance, incomplete stories	67
7.1	Representation accuracy	88
8.1	Feasibility of retrieval	107
9.1	Concept categories	112
9.2	Accuracy of DISCERN lexicon	121
10.1	Paraphrasing performance in isolation	129
10.2	Question answering performance in isolation	129
10.3	Paraphrasing performance in chain	130
10.4	Question answering performance in chain	130
10.5	Paraphrasing performance with incomplete stories	131
10.6	Question answering performance with incomplete stories	131
11.1	Reducing complexity with hierarchies	165
11.2	Effect of FGREP on training time	165

11.3 Effect of more modules on training time 166

ACKNOWLEDGMENTS

Special thanks go to Michael Dyer for continuous support, many enthusiastic discussions and excellent suggestions on earlier papers and drafts of this dissertation. I would also like to acknowledge committee members Josef Skrzypek for constructive criticism on the plausibility of the research, claims and terminology, Jacques Vidal for initially pointing me to the feature maps, Eric Halgren and Patricia Greenfield for providing interdisciplinary pointers and criticism. UCLA and especially the Artificial Intelligence Laboratory formed a stimulating environment for research. Special thanks go to Trent Lange and Othar Hansson for useful comments on an earlier draft of this thesis and for Bruce Abramson for lending me his MacIIci to prepare the manuscript on. I would also like to acknowledge Teuvo Kohonen and Jay McClelland for their support and several inspiring discussions. Figure 12.1 is included with permission from Mark St. John.

The research was supported in part by ITA Foundation, W. M. Keck Foundation, Thanks to Scandinavia Foundation, and the Hewlett-Packard equipment grant for Artificial Intelligence research, and in part by the Academy of Finland, Emil Aaltonen Foundation, Foundation for the Advancement of Technology, Alfred Kordelin Foundation, the Finnish Science Academy, Finnish Cultural Foundation, Jenny and Antti Wihuri Foundation, and the Economic and Technological Sciences Foundation.

VITA

- Dec. 16, 1961 Born, Helsinki, Finland
- 1984 B.A., Mathematics/Computer science
Southwestern University, Georgetown, Texas
- 1984–1986 Research and teaching assistant
Department of Mathematics, Helsinki University of Technology
Helsinki, Finland
- 1986 M.S., Systems analysis and operations research
Helsinki University of Technology
Helsinki, Finland
- 1987–1990 Research assistant
Computer Science Department
University of California, Los Angeles

PUBLICATIONS

- Miikkulainen, R. (1987). Self-organizing process based on lateral inhibition and weight redistribution. Technical Report UCLA-AI-87-16, Artificial Intelligence Laboratory, Computer Science Department, University of California, Los Angeles.
- Miikkulainen, R. and Dyer, M. G. (1987). Building distributed representations without microfeatures. Technical Report UCLA-AI-87-17, Artificial Intelligence Laboratory, Computer Science Department, University of California, Los Angeles.
- Miikkulainen, R. and Dyer, M. G. (1988). Forming Global Representations with Extended Backpropagation. In *Proceedings of the IEEE Second Annual International Conference on Neural Networks (ICNN-88)*. IEEE.
- Miikkulainen, R. and Dyer, M. G. (1989). Encoding input/output representations in connectionist cognitive systems. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.

- Miikkulainen, R. and Dyer, M. G. (1989). A modular neural network architecture for sequential paraphrasing of script-based stories. In *Proceedings of the International Joint Conference on Neural Networks*. IEEE.
- Miikkulainen, R. and Dyer, M. G. (1990). Natural Language Processing with Modular Neural Networks and Distributed Lexicon. Technical Report UCLA-AI-90-02.
- Miikkulainen, R. (1990). A PDP architecture for processing sentences with relative clauses. In *Proceedings of the 13th International Conference on Computational Linguistics*.
- Miikkulainen, R. (1990). A distributed feature map model of the lexicon. In *Proceedings of the Twelfth Annual Cognitive Science Society Conference*.
- Miikkulainen, R. (1990). A neural network model of script processing and memory. Technical Report UCLA-AI-90-03.
- Miikkulainen, R. (in press). Script recognition with hierarchical feature maps. *Connection Science*.

ABSTRACT OF THE DISSERTATION

DISCERN: A Distributed Artificial Neural Network Model of Script Processing and Memory

by

Risto Pekka Miikkulainen

Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1990
Professor Michael G. Dyer, *Chair*

The dissertation has three main goals: (1) to show that a complete natural language processing (NLP) system can be built from distributed artificial neural networks, (2) to show that several high-level phenomena can be explained at the physical level using the special properties of these networks, and (3) to address various technical issues in connectionist cognitive modeling by developing new network architectures and techniques.

DISCERN is a large-scale NLP system implemented entirely at the subsymbolic level. It learns to read short narratives about stereotypical event sequences, store them in episodic memory, generate fully expanded paraphrases of the narratives, and answer questions about them. The system input and output consists of sequences of orthographic word symbols, i.e. DISCERN processes sequential natural language.

Several high-level phenomena emerge automatically from the special properties of distributed neural networks. DISCERN learns to infer unmentioned events and unspecified role fillers, and it generates expectations and defaults and exhibits plausible lexical access errors and memory interference behavior. Word semantics, memory organization and the appropriate script inferences are automatically extracted from examples.

Processing in DISCERN is based on hierarchically-organized backpropagation modules, communicating through a central lexicon of word representations. The lexicon is a double feature map, which transforms the orthographic word symbol into its semantic representation and vice versa. The episodic memory is a hierarchy of feature maps, where memories are stored "one-shot" at different locations. Special mechanisms were developed for modular training, automatically developing distributed representations, role binding, type/token processing, lexical disambiguation, sequential communication, filtering out internal and external noise, many-to-many mapping, hierarchical self-organization and one-shot storage.

Part I

OVERVIEW

Chapter 1

Introduction

1.1 Task

DISCERN (DIstributed SScript processing and Episodic memoRY Network) is a distributed artificial neural network system which learns to process simple stereotypical narratives. To see what DISCERN is up against, let us consider the following examples:

- (1) John went to MaMaison. John asked the waiter for lobster. John left a big tip.
- (2) John went to LAX. John checked in for a flight to JFK. The plane landed at JFK.
- (3) John went to Radio-Shack. John asked the staff questions about CD-players. John chose the best CD-player.

The first narrative mentions only three events about John's visit to MaMaison. Because restaurant visits are common experiences with highly regular events, a human reader immediately assumes a number of events which certainly or most likely must have occurred. For example, he assumes that the waiter seated John, that John ate the lobster, that John paid the waiter etc. Based on the fact that the restaurant serves lobster, and that John left a big tip, the reader can also guess that the restaurant was probably of the fancy type rather than a fast-food type; the food was probably good, etc. If asked to elaborate, the reader might come up with the following expanded paraphrase:

John went to MaMaison. The waiter seated John. The waiter gave John the menu. John asked the waiter for lobster. John waited for a while. The waiter brought John the lobster. John ate the lobster. The lobster tasted good. John paid the waiter. John left a big tip. John left MaMaison.

Or, the reader could answer questions about the stories in the following way:

- Q: What did John buy at Radio-Shack?
A: John bought a CD-player at Radio-Shack.
Q: Where did John take a plane to?
A: John took a plane to JFK.
Q: How did John like the lobster at MaMaison?
A: John thought the lobster was good at MaMaison.

We can identify a number of issues from the above examples. The answers and the paraphrase show that the reader makes a number of inferences beyond the original story [Schank and Abelson, 1977]. The inferences are not based on specific rules but on statistical regularities, learned from experience. The reader has experienced a number of similar event sequences in the past and the unmentioned events and properties have occurred in most cases. They are assumed immediately and automatically, and quickly become part of the memory about the narrative. If the reader is asked later whether John actually flew to JFK, he would quite confidently confirm this, but might not be able to tell whether this event was actually mentioned in the story or only inferred [Bower et al., 1979].

Another group of issues concerns episodic memory. Narratives are stored in memory one at a time as they are read in, with only a single presentation. We don't usually have to go back and re-activate all previous narratives after we have read a new story. However, the new story is recognized as an instance of a familiar sequence of events, and attention is paid only to the facts that are specific to this story. It seems that episodic memory is structured to support classifying according to the similarities and storing the differences, and this structure has been extracted from experience [Kolodner, 1984].

Episodic memory structure also supports associative retrieval. A question supplies only partial information about the story it is referring to, yet the story is retrieved with only the question as a cue. The more unique the story is in the memory, the less needs to be specified in the question. If there is only one travel story, we can unambiguously ask *Where did John travel to?* Usually, if there are several alternative stories, the most recent one is recalled by default. Context of the previous question can also be used to select among the alternatives [Lehnert, 1978]. And of course, it is possible to recognize a situation where there is nothing appropriate in the memory.

Issues concerning the lexical and semantic memory of DISCERN are perhaps less obvious. The system must be able to represent word meanings, otherwise we cannot say it is performing any kind of cognitive processing. The meanings could be coded in by hand, using some elaborate representation scheme. However, it would be preferable to *learn* the meanings from examples. The properties of the words which are most crucial to the task should be extracted and coded into the representation automatically. Finally, a mechanism for mapping the lexical word symbols to these meanings is necessary. This is a many-to-many mapping, since some lexical words correspond to several concepts, and the same concept can usually be expressed with several different words.

1.2 Motivation and goals

Understanding stories about stereotypical event sequences is not a new task. Script theory in cognitive science was developed to explain how knowledge about familiar everyday routines is used in story understanding [Schank and Abelson, 1977]. The theory was developed along with symbolic models which implemented the theory at various levels [Cullingford, 1978; DeJong, 1979]. These models are capable of quite impressive behaviour in their domains and certainly capable of processing simple script instantiations such as above examples.

What makes this task worth doing with artificial neural networks? There are several

questions that the symbolic approach does not address. The processing architecture, mechanisms and knowledge in symbolic systems are hand-coded with a particular domain and data in mind. Inferences are based on handcrafted rules and representations of the scripts. Such systems cannot utilize the statistical properties of the data to enhance processing; they can only do what their creator explicitly programmed them to do. This contradicts the intuitive notion of scripts. They should not be preset, fixed, all-or-none representations, but instead emerge automatically from the statistical regularities in the experience.

The symbolic models are high-level process models, far removed from the physical structures that implement the processes in the brain. As a result, they inherently lack the capability of explaining certain aspects of human performance. They cannot address questions such as: Where do performance errors come from? How can memory become overloaded and why do certain types of memory confusions occur in overload situations? What happens when the system is corrupted with noise, or when parts of it are destroyed?

A major motivation for DISCERN is to give a better account of the high-level cognitive phenomena outlined above, in terms of the special properties of distributed neural networks such as learning from examples, automatic generalization and graceful degradation. Specifically, DISCERN aims at showing how script-based inferences can be learned from experience, based on the statistical correlations in the example data, how the episodic memory organization can be automatically formed based on the regularities in the example data, and how word semantics can be learned from examples of their use. The model should give plausible explanations on how expectations and defaults automatically emerge. Because artificial neural network models are motivated by the physical structures in the brain, they stand a chance of better explaining performance errors and deficits. DISCERN should exhibit plausible memory interference errors, role binding errors and lexical errors in overload situations and in the face of noise and damage.

On the other hand, DISCERN is a demonstration that a large-scale natural language processing (NLP) system, which performs at the level of symbolic NLP models, can be built from distributed artificial neural networks. A script processing system needs to read its input stories word by word in natural language. It needs to process meaningful internal representations where all inferences are explicit, so that its behaviour is interpretable to the external observer. It needs to have long-term memories both for the general, statistical knowledge about scripts and word semantics, and for the specific, exact knowledge about actual stories with actual word instances. The model should be able to produce fully expanded paraphrases of the stories, including all inferences. It should be able to answer questions about specific events of a specific story in the memory. Finally, the output should be produced word by word in natural language.

Script understanding forms an excellent testbed for research in neural network techniques. Much of the research in network architectures and algorithms has proceeded without a clear motivation at higher levels, and often the results do not provide solutions to important problems in cognitive modeling. We believe that by using a concrete high-level problem to identify technical issues and to motivate solutions to them, we can develop techniques which address real problems in the task. This approach also makes us well aware of the limitations of these techniques.

The third major goal of DISCERN is to develop general network mechanisms for dealing with the complexity and structure of higher-level cognitive modeling. These include modular network architectures, automatic learning of distributed representations, methods for one-shot learning, hierarchical self-organization, many-to-many mappings, role binding and type/token processing. These techniques are designed to solve problems in script processing, but should prove useful in other high-level tasks as well.

1.3 Approach

Previous research in parallel distributed processing (PDP) has concentrated mostly on isolated, small, low-level tasks, and relied heavily on pre- and postprocessed data [Sejnowski and Rosenberg, 1987; Rumelhart and McClelland, 1987; Rumelhart et al., 1986c]. In many cases, the problem is reduced to learning a simple mapping. As a result, such PDP models typically have very little internal structure or architectural complexity. They produce the statistically most likely answer given the input conditions, in a process which is opaque to the external observer. This approach is well suited for modeling isolated low-level tasks, such as learning past tense forms of verbs or pronunciation of words. However, modeling higher-level cognitive tasks with homogeneous networks has been unfeasible, for three reasons: (1) Higher-level tasks usually require composite subtasks, i.e. they consist of several distinct sub-processes, such as parsing language, generating language, memory storage, memory retrieval, reasoning etc., which cannot be performed in a single pattern transformation. Complex behaviour requires bringing together several different kinds of knowledge and processing, which is not possible without structure [Simon, 1981; Feldman, 1989]. (2) The required network size, the number of training examples and the training time becomes intractable as the size of the problem grows, especially in sequential processing [Servan-Schreiber et al., 1989; Harris and Elman, 1989]. (3) There is no way to evaluate what the entire system is doing, what knowledge it is acquiring, applying etc. unless each module processes meaningful internal representations which can be interpreted by external observers, and by other modules in the system.

A plausible approach for higher-level cognitive modeling, therefore, is to construct the architecture from several interacting modules, which work together to produce the higher-level behaviour [Minsky, 1985]. Central issues to be addressed in this approach are: (1) how the overall task should be broken into modules and how the modules should be organized, (2) what are the appropriate architectures for different kinds of subtasks, (3) how the sub-networks should be designed so that they can serve as modular building blocks, (4) how communication between such building blocks can be established, and (5) how to make use of the modular structure in training the system.

Any complex architecture will need to have a common set of terms, serving the function of a "symbol table" for intercommunication. In a large system consisting of many modules, with many communicating pairs, the most efficient way to establish this function is through a global vocabulary, a central lexicon. The modules communicate using terms from this global symbol table, instead of having a separate set of terms for each communication channel. Each module can then be trained separately and in parallel, as long as they are trained with

compatible I/O data.

In a high-level PDP model, communication (i.e. input/output of each module) can take place using distributed representations. In backpropagation networks with hidden layers, the network automatically develops internal distributed representations for the input/output items as a side effect of learning the processing task [Rumelhart et al., 1986b; Hinton, 1986; Miikkulainen and Dyer, 1987; Elman, 1990]. These representations reflect the regularities of the task and data, extracted without external supervision, and provide an excellent basis for generalization. Central to this thesis is a mechanism for forming distributed representations, called FGREP [Miikkulainen and Dyer, 1987; Miikkulainen and Dyer, 1989a], where the internal representations discovered by the network are made public in a global lexicon, so that they can be used for communication in a large modular system.

To serve as a modular building block, the input and output of a network must be self-contained and publicly interpretable, so that other modules can be trained to use the same data. The module has to represent its processing knowledge explicitly in its output. The FGREP architecture was designed according to this principle, and serves as the basic processing building block for DISCERN. Hierarchical organization of FGREP modules with a sequential interface is well suited for language processing tasks, and also turns out to be a very efficient way to reduce the complexity of the task.

However, backpropagation-based network architectures do not lend themselves well to modeling long-term memory. They can represent ambiguity only by blending the alternatives, whereas memory ambiguity often requires representing the alternatives simultaneously and distinctively. Also, backpropagation networks cannot be trained with single presentations of items. The whole set of possible inputs must be presented multiple times to prevent unlearning other items [Rumelhart et al., 1986b].

The memory modules in DISCERN are implemented with feature maps. Distributed representations of the data are laid out spatially over a 2-D area, assigning different locations to different items. This approach makes many-to-many mappings and one-shot learning possible. A hierarchical structure of several maps, together with the spatial organization of each component map, provides organization for the data and results in several plausible memory properties.

The processing and the memory modules in DISCERN can be trained separately and in parallel, making good use of the modular architecture. Their tasks appear completely independent during training, but the training data ties them together. The networks are trained with I/O data based on the same set of script-based stories, with the same set of representations in the central lexicon. They learn compatible tasks, and perform well together when connected.

1.4 Guide to the reader

The next chapter contains the necessary background material. The basics of script theory are reviewed and the unique qualities of the parallel distributed processing approach are outlined and contrasted with the symbolic approach. *Chapter 3* gives an overview of the

DISCERN architecture. The overall modular system structure is presented at a block-diagram level, and the design and function of each module is described in general terms. This chapter provides the framework and perspective for the detailed discussion of the specific techniques in parts II and III.

In part II of the thesis, the mechanisms related to the backpropagation-based processing modules are discussed in detail. The technique for forming global representations with extended backpropagation (FGREP) is introduced in *chapter 4* in the context of sentence processing, i.e. in the task of assigning case roles to sentence constituents. The properties of the FGREP representations are discussed at length. We show how the vocabulary can be extended by creating several distinct representations with the same properties (e.g. John, Mary, Bill from the word human), i.e. several tokens from the same type. The FGREP-method is then extended to sequential input and output. The recurrent FGREP network can perform case-role assignment from word-by-word natural language input. *Chapter 5* shows how a complex system can be built from hierarchically organized recurrent FGREP modules, together with a central lexicon of words. Detailed script processing performance and learning results for the processing subsystems of DISCERN, i.e. DISCERN without memory, are presented.

Part III concentrates on the memory mechanisms. *Chapter 6* gives background in self-organizing feature maps and why they are well-suited for modeling memory. The discussion of episodic memory begins with the hierarchical feature mapping technique (*chapter 7*), which is used to self-organize the memory according to the implicit structure in the example data. Creation of the actual memory traces is implemented with the trace feature map technique, the topic of *chapter 8*. Up to this point in the discussion, the lexicon has been treated as a black box containing the word representations. The box is opened in *chapter 9*, and the double feature map structure of the lexicon is revealed. The capability of the lexicon to model dyslexic and aphasic impairments is also discussed.

Part IV brings the different parts of DISCERN together again for an evaluation of the complete model. *Chapter 10* presents overall performance statistics and annotated traces of several interesting I/O cases. The successes and shortcomings of the model are evaluated in *chapter 11*. The research is contrasted to previous work in related areas in *chapter 12*, and future directions and possibilities are outlined in *chapter 13*.

Appendix A contains a transcript of the story data used in training and testing the system and its modules. Details of the hardware and software used in implementing DISCERN are given in *appendix B*. In *appendix C*, the central parts of the training programs for the different components of DISCERN are listed, and the necessary data files are described. The core of the computer program that makes up DISCERN in the performance phase is listed in *appendix D*, together with descriptions of the data files. Code and data for the various analysis and comparison routines in this dissertation are presented in *appendix E*.

Chapter 2

Background

2.1 Scripts

According to script theory, people organize knowledge of everyday routines in the form of scripts [Schank and Abelson, 1977]. Scripts are schemata of often-encountered, stereotypical sequences of events. Common knowledge of this kind makes it possible to efficiently perform social tasks such as visiting a restaurant, visiting a doctor, shopping at a supermarket, traveling by airplane, attending a meeting, etc. People have hundreds, perhaps thousands of scripts at their disposal. Each script divides further into different variations, or tracks. For example, there is a fancy-restaurant track, fast-food track and a coffee-shop track for the restaurant script, each with slightly different events.

Script theory is attractive because it accounts for a fairly well defined, manageable part of cognition. Scripts are intuitively plausible and well supported by experimental evidence [Graesser et al., 1979; Sanford and Garrod, 1981; Sharkey and Mitchell, 1985; Sharkey and Sharkey, 1987]. For example, Bower, Black and Turner conducted a number of studies about scripts in text processing [Bower et al., 1979]. They found that people in general were in good agreement on the characters, props, actions and their order in the scripts. Subjects also tended to confuse the actions which were mentioned in the story with the actions which were simply inferred from the script. Statements about events which immediately follow each other in the script were read faster than statements which were further apart.

Scripts also provide a computational theory, similar in spirit to the frame theory of representation in artificial intelligence [Minsky, 1981]. The basic idea behind both is schematic knowledge: people organize knowledge about familiar objects, events and situations in terms of prototypes, or schemata. A schema contains representation for common knowledge, shared by all instances, and a number of slots which take different values for different instances. For example, a schema for a room contains the common structure (walls, floor, ceiling, door), and slots for size, shape, furniture, etc. A schema for a restaurant visit contains the common events (entering, seating, ordering, eating, etc.) and slots for the restaurant, food, amount of the tip, etc. (table 2.1).

Knowledge of scripts is important in natural language understanding because it allows for efficient communication. In narrative texts, many details are left out, because they can easily be filled in by the reader using scriptal knowledge. For example, in reading:

John went to fancy-restaurant. John asked the waiter for lobster.
John left a big tip.

the reader understands the last sentence by assuming the usual intermediate events in the restaurant script: the waiter brought John the lobster, John ate the lobster, the lobster

RESTAURANT SCRIPT		
FANCY-RESTAURANT TRACK		
Causal Chain:	Roles:	
Entering	Customer	= John
Seating	Restaurant	= MaMaison
Ordering	Food	= lobster
Eating	Taste	= good
Paying	Tip	= big
Tipping		
Leaving		

Table 2.1: Representation of a script-based story as a causal chain and role bindings.

tasted good, John paid the waiter, and John left a big tip (because the food/service was good). More complex texts contain references to several scripts, and understanding requires recognizing and applying several scripts at the same time. Scripts are used to connect the pieces of the text together, i.e. they are tools in building a complete representation of the story.

In symbolic NLP a script is represented as a causal chain of events with a number of open roles (table 2.1) [Schank and Abelson, 1977; Cullingford, 1978; DeJong, 1979; Dyer et al., 1987]. Applying this knowledge to a story requires identifying the relevant script and filling in its roles with the actual constituents of the story. Once the script has been recognized and the roles have been instantiated, the sentences are matched against the events in the script. Events which were not mentioned in the story but are part of the causal chain can be inferred, as well as certain fillers of roles which were not specified in the story. For example, in the above story it is plausible to assume that John ate the lobster and the lobster tasted good. Understanding is usually demonstrated by generating an expanded paraphrase of the original story and by answering questions about it.

DISCERN does not aim at further development of the psychological theory of scripts. The theory is used as a basis for modeling more or less as it is. Neither does the system implement the complete theory or the full complexity of symbolic script processing systems. The narratives in our system are limited to simple script-based stories, i.e. instantiations of the event sequence of a single script with its specific role bindings. The major contributions of DISCERN are in showing how script processing can be learned from examples by a neurally inspired architecture.

2.2 Parallel distributed processing

There is fairly convincing evidence that knowledge is represented distributively in the brain [Lashley, 1950; Kosslyn and Hatfield, 1984]. Memories or particular activities may be located in limited regions, but there are several memories represented within the same region, and destruction of a part of a region often does not result in a complete loss of any particular

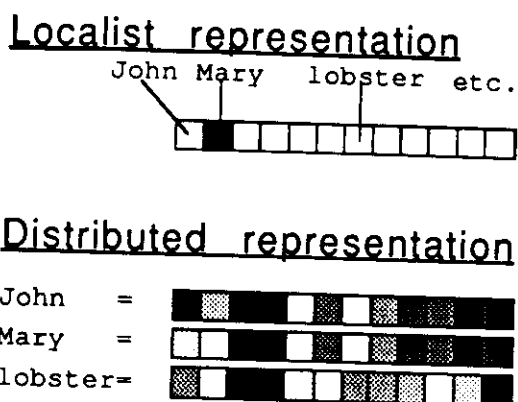


Figure 2.1: **Localist and distributed representations.** In the localist representation, each unit stands for a particular item. The number of items is directly limited by the number of units, but several items can be represented in parallel. In the distributed representation, each item is represented as a pattern of activity over the same set of units. The number of items that can be represented depends on the accuracy of the unit activity values. Only one item can be represented at a time, but blends of representations are possible. The values are indicated by gray-scale coding in the figure.

function. It seems that information is spread out over the whole area, with different pieces of information superimposed, and that each part of the region is equipotential in generating the information in case of partial damage.

Parallel distributed processing (PDP) models have recently emerged as an alternative to symbolic modeling in cognitive science. While they do not always aim at direct biological plausibility, the approach is motivated by the distributed representation in the brain. PDP systems consist of a large number of densely interconnected simple processing units, or artificial, highly abstracted “neurons”. A single unit does not represent any particular item, nor does a single connection weight stand for any particular relation. Each unit and connection is involved in representing many different pieces of information. The data items are represented as *different patterns of activity* over the same set of units (figure 2.1), and the processing knowledge for different situations is superimposed over the same set of connections.

The distributed approach is fundamentally different from the symbolic approach [McClelland et al., 1986; Hinton et al., 1986; Derthick and Plaut, 1986; van Gelder, 1989], and consequently, also from the localist neural network models (see [van Gelder, 1989]). In fact, the distributed and symbolic approaches are incompatible. Symbolic representations are disjoint and discrete, highly grammatical and compositional. It is always possible to tell whether a symbolic token belongs to a certain symbol class, and it is possible to change the symbolic tokens in a composite symbolic representation without affecting the rest of the representation. On the other hand, distributed representations are typically continuous, non-grammatical and non-concatenative [van Gelder, 1989]. There are no clear-cut distributed classes, because representations belong to each and every class to a different degree, depending on their similarities to other representations. Addition of another item in distributed memory affects the representations of all other items. Items are only represented in the

context of all other items in the memory.

For this reason, distributed systems are not just lower-level implementations of the symbolic cognitive models. It is not possible to *implement* the symbolic level with an incompatible approach [van Gelder, 1989]. The whole of cognitive science must be built from the distributed level up, and the result is conceptually independent from symbolic cognitive science.

However, it is quite plausible and most useful to build systems which *behave* as if they were manipulating symbolic structures at the high level. Much of human reasoning appears symbolic, although the brain cannot have strictly symbolic representations. The content of the information structures can be similar to the symbolic systems, although the structures themselves are different. In other words, even if PDP is incompatible with the symbolic approach, it is still possible to use the results of symbolic research to guide research in building PDP models of cognition.

Several interesting processing enhancements arise from the distributed approach. These are based on four basic properties: (1) the representations are continuously valued, (2) similar concepts have similar representations, (3) several different pieces of knowledge are superimposed on the same limited hardware, each memory affecting each other, and (4) the representations are holographic, i.e. any part of the representation can be used to reconstruct the whole.

The first two properties result in the representations reflecting the meanings of the concepts they stand for. Because they are continuous, it is possible to represent shades of meaning and category memberships become a matter of degree. The third property gives us spontaneous generalization. When several representations are superimposed in the same memory, the individual variations tend to cancel out, and regularities get enhanced in the memory. Knowledge about some class of items is automatically generalized into all similar items. The system can perform in novel situations, based on the regularities it has discovered in its previous experience. The third property also results in graceful degradation when overloading the memory. There is no hard limit on how much can be stored, but the representations become less and less accurate.

The holographic property makes the system robust against noise and damage. Noise in the input patterns is effectively filtered out, because random deviations tend to cancel out. Loss of some processing elements is not crucial, because the information is available elsewhere. Instead of selectively losing information, the performance degrades gracefully. For the same reason, incomplete input patterns can be automatically completed. As a result, distributed systems are inherently content-addressable. This property can be used at the high level to automatically create defaults and expectations.

Perhaps the greatest advantage of the distributed approach is the abundance of simple learning mechanisms which allow the processing and representation of information to be automatically extracted from examples. Some of these mechanisms require an external trainer, who tells the network what the correct output is for each input [Rumelhart et al., 1986b], others only need reinforcement information without an exact target [Barto et al., 1981], and certain algorithms work completely unsupervised, self-organizing their parameters into an internal statistical model of the data [Rumelhart and Zipser, 1986; Grossberg, 1987; Ko-

honen, 1984]. The same network architecture can learn to process a wide variety of inputs and take advantage of the implicit statistical regularities in the data, without having to be specifically programmed with particular data in mind [McClelland et al., 1986]. The gradual evolution of the system performance as it is learning often resembles human learning in the same task [Rumelhart and McClelland, 1987; Sejnowski and Rosenberg, 1987].

However, parallel distributed processing in homogenous networks alone is not sufficient for high-level cognitive modeling. A major shortcoming is that they are *not parallel at the knowledge level*. PDP models can represent multiple simultaneously active items only by blending them [Sumida and Dyer, 1989]. As will be discussed under the memory mechanisms of DISCERN, it is often necessary to complement PDP with a degree of localization, by laying out different information on different parts of the hardware. There is also considerable neurological evidence for certain specific cognitive impairments resulting from localized brain damage (see section 9.5.2).

Processing modules in DISCERN employ backpropagation learning, a supervised algorithm which is described in section 4.1. The DISCERN memory modules are based on self-organizing feature maps (section 6), which employ unsupervised learning to form a spatial layout of the memory data.

Chapter 3

Overview of DISCERN

DISCERN processes script-based narratives which are instantiations of the causal chains of events with different role bindings. For example, choosing customer=John, restaurant=MaMaison, food=lobster gives us one fancy-restaurant story, while selecting Bill, Leone's and steak would give a different one. For this discussion, DISCERN was trained with three different scripts, with three tracks each: restaurant (fancy, coffee-shop and fastfood tracks), shopping (clothing, electronics and grocery tracks) and travel (plane, train and bus tracks).

The causal chain of events in a script is a summary of all restaurant experiences and remains stable, whereas the role bindings are different in each application of the script. This distinction suggests a neural network approach for processing scripts. The causal chain of events is learned from exposure to a number of restaurant stories, and is stored in the long-term memory of the network, i.e. in the weights. The role bindings are processed as activity patterns in data-specific assemblies.

3.1 System architecture

A block diagram of DISCERN is shown in figure 3.1. The model can be divided into four subsystems: parsing, generating, question answering and memory. Each subsystem consists of two modules. The modules are trained in their respective tasks separately and in parallel. During performance, the modules form a network of networks, each feeding its output to the input of another module.

The parsing, generating and question answering modules perform the processing in the system. These modules are either simple recurrent or feedforward networks, and they are trained with backpropagation. The memory modules are implemented with feature maps. The input and output of DISCERN consists of distributed representations of orthographic word symbols. Internally, DISCERN processes distributed representations of semantic concepts.

3.1.1 Lexicon

The memory subsystem consists of two modules: the episodic memory and the lexicon. The lexicon stores the lexical symbol and semantic concept representations and translates between them (figure 3.2). The lexicon consists of two parts:

- (1) **The lexical memory** stores the distributed representations for the physical word symbols. In DISCERN, they stand for the visual properties of the orthographic lexical symbols. These symbols are a property of the environment, and they are preset and fixed

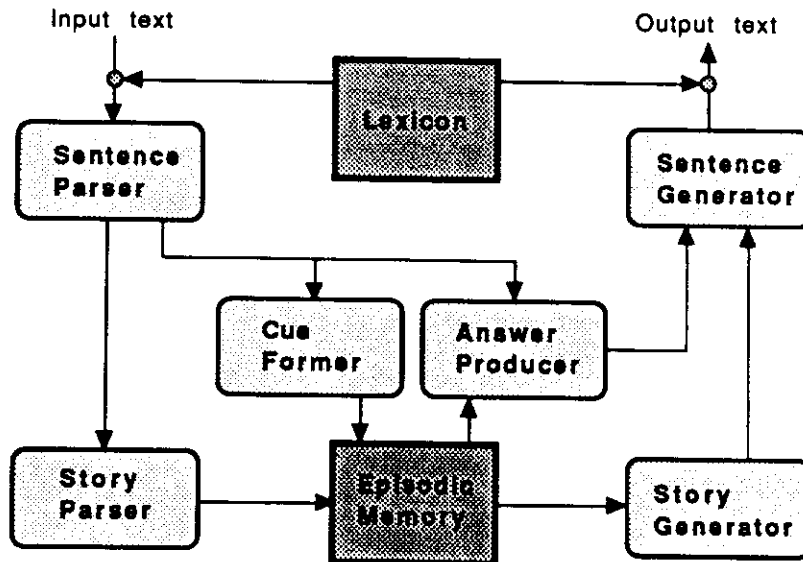


Figure 3.1: Block diagram of DISCERN (performance phase). The model consists of parsing, generating, question answering and memory subsystems, two modules each. A dark square indicates a memory module, a light square indicates a processing module.

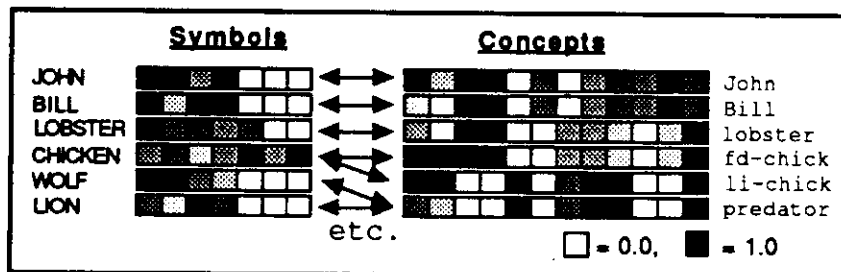


Figure 3.2: Lexicon. The lexicon stores the distributed representations of lexical symbols and semantic concepts. Each representation is a vector of real numbers between 0 and 1, shown as grey-scale values from white to black in the figure.







Agent	Act	Recipt	Pat-attr	Patient	Location	Case roles
John	left	waiter	big	tip		Concepts
						Concept representations

Figure 3.3: Case-role representation of the sentence John left the waiter a big tip. The concept representations in each case-role correspond to the concept representations in the lexicon.

throughout the learning and performance of DISCERN. These representations are referred to as symbol representations, or symbols, in the text and they are labeled with UPPERCASE COURIER typeface.

(2) **The semantic memory** contains representations for semantic concepts. These representations stand for distinct meanings. They are formed automatically during training, based on how the concepts are used in the training examples. The representations reflect the semantics of the concepts. Concepts which are used in similar ways in the data have similar representations. These representations are referred to as concept representations (concepts), but also simply as words in the following discussion. They are labeled with lowercase courier typeface.

The lexicon acts as a filter in two places in DISCERN: at the input it translates each orthographic input symbol in the input text into the corresponding concept, and passes it on to the parsing subsystem as input. At the output, it translates each concept representation produced by the generating subsystem into the corresponding orthographic symbol, and passes it on to the stream of output text.

3.1.2 Parsing subsystem

The parsing subsystem reads the sequence of concept representations into the internal representation of the story. The subsystem consists of two hierarchically organized modules:

(1) **The sentence parser** reads each input sentence sequentially word by word and forms an assembly-based case-role representation of the sentence as its output (figure 3.3). The case-roles are based on those of [Fillmore, 1968]. The representation is essentially a vector with partitions for the case roles. Each partition is filled with the distributed representation for the concept that fills that role.

(2) **The story parser** reads these case-role representations one at a time, and forms an assembly-based representation of the whole story as its output (figure 3.4). This is again a vector with partitions for the roles in the script, with two extra assemblies indicating the script and the track. The partitions stand for different roles depending on the pattern in the script-assembly. For example, in the restaurant script the fifth assembly indicates the restaurant, while in the travel script it stands for the destination. This data-specific slot-filler representation (in terms of the script, track and the role bindings) constitutes the internal representation of the story. All inferences about missing events and missing fillers are completed by the story parser and coded into the story representation. The original

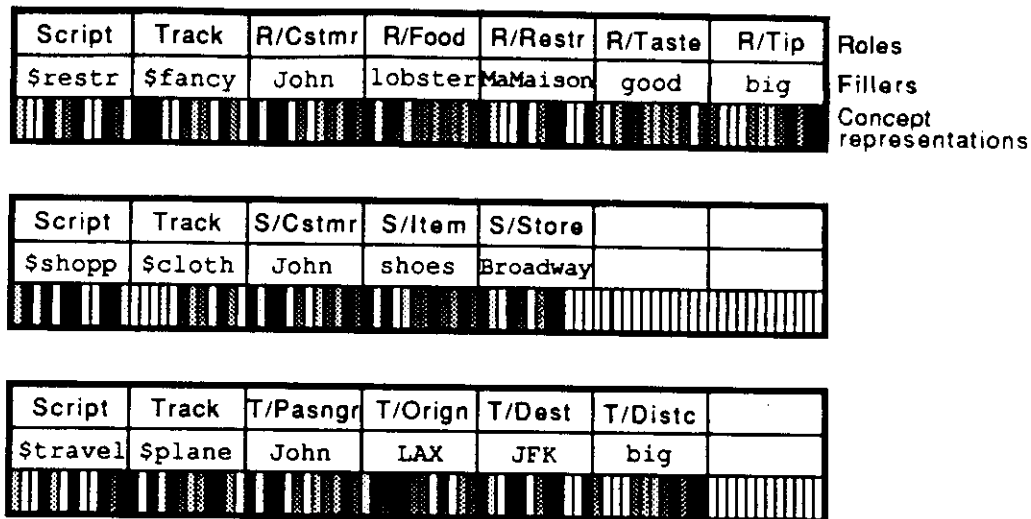


Figure 3.4: Representation of stories by their role bindings. The assemblies are data-specific: their interpretation depends on the pattern in the script and track slots. The role names R/... are specific for the restaurant script, S/... for the shopping script and T/... for the travel script.

events are no longer available; the representation is all that remains of the original story.

3.1.3 Episodic memory

Episodic memory has the task of storing and retrieving the internal story representations. During training, the memory forms a hierarchical taxonomy of script-based stories in a self-organizing process. The taxonomy provides organization for the memory. During performance, each input story representation is classified as an instance of a script, track and role binding, and the appropriate location in the memory is found (figure 3.5). A trace is created at that location "one-shot", with a single presentation, possibly interfering with other previous traces of similar stories.

The memory can later be cued with a partial representation of a story, and the retrieval mechanism finds the most similar trace in the memory, if there is any. A complete representation of that story is then returned. In other words, the episodic memory functions as a content addressable associative memory for stories.

3.1.4 Generating subsystem

The story generator and sentence generator modules constitute the generating subsystem. Their task is to generate a full paraphrase of the story from its internal representation. This process is simply the reverse of reading. The story generator takes the internal representation as its input, and produces the case-role role representations of the sentences one at a time as its output. These are input to the sentence generator, which generates the words of each sentence one at a time.

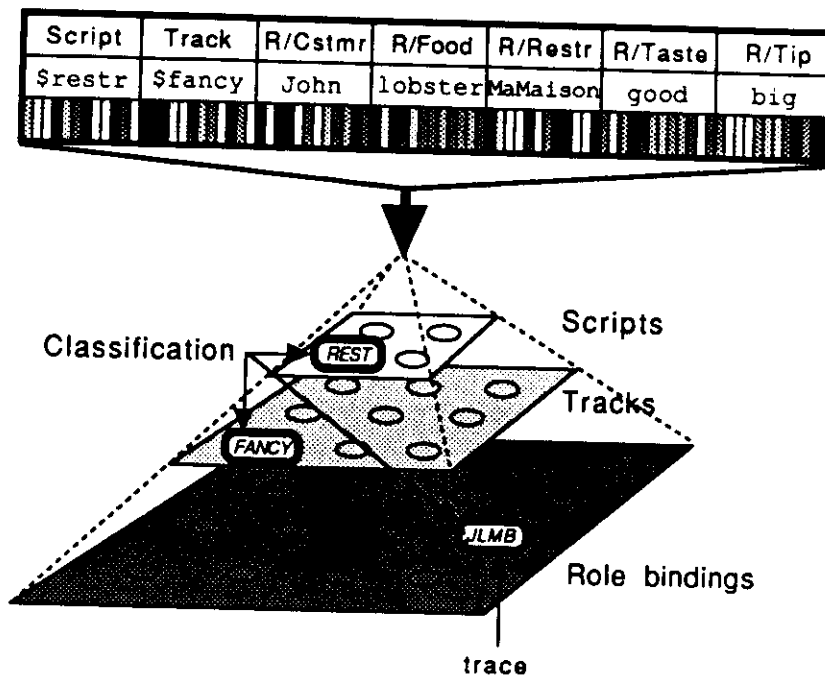


Figure 3.5: **Episodic memory.** The memory is organized according to the hierarchical taxonomy of script-based stories. An input story representation is classified as an instance of a particular script, track and role binding. The classification specifies a unique location for the story at the bottom level, and a trace is created at that location. JLMB stands for a fancy-restaurant role binding customer=John, food=lobster, restaurant=MaMaison and tip=big/taste=good.

Agent	Act	Recipnt	Pat-attr	Patient	Location
John	ate			what	MaMaison




Figure 3.6: **Case-role representation of the question** What did John eat at MaMaison? Questions are represented as sentences, but processed through different pathways.

Script	Track	R/Cstrmr	R/Food	R/Restr	R/Taste	R/Tip
\$restr	\$fancy	John	?	MaMaison	good	?




Figure 3.7: **Memory cue.** Most of the story representation is complete, but the patterns in Food, Taste and Tip slots indicate averages of all possible alternatives.

3.1.5 Question answering subsystem

A question is read in sequentially word by word just like any sentence in a story, and the sentence parser forms a case-role representation for it (figure 3.6). This is input to **the cue former** module, which produces a partial representation of the appropriate story as its output (figure 3.7). For example, if the question is **What** did John eat at **MaMaison**?, we know that the question is referring to a fancy-restaurant visit (because **MaMaison** is a fancy restaurant), where John was the customer, **MaMaison** was the restaurant and the food was good (it is always good in fancy-restaurants in our data). The rest of the assemblies are filled with a best guess, i.e. the average of all alternatives. The partial representation is then fed as a cue to the episodic memory, which completes it.

The answer producer module takes both the question and the completed story representation as its input, and produces the case-role representation of the answer as its output (figure 3.6). This is translated into output text by the sentence generator. Together the cue former and answer producer constitute the question answering subsystem of the model.

3.2 I/O example

Below is a listing of DISCERN output as it processes the above example story. Comments printed out by DISCERN are enclosed in brackets, otherwise all words stand for activity patterns, with “_” indicating a blank pattern. Output layers consisting of several assemblies are enclosed between bars.

Agent	Act	Recipnt	Pat-attr	Patient	Location
John	ate		good	lobster	




Figure 3.8: **Case-role representation of the answer** John ate a good lobster.

[parsing input story:] (a comment by DISCERN)
 JOHN WENT TO MAMAISON . (sequence of input words)
 |John went - - - MaMaison| (sentence parser output)
 JOHN ASKED THE WAITER FOR LOBSTER .
 |John asked waiter - lobster -|
 JOHN LEFT A BIG TIP .
 |John left waiter big tip -| (see figure 3.3)

[into internal rep:] (story parser output
 |\$restaurant \$fancy John lobster MaMaison good big|
 see figure 3.4)

[storing into episodic memory:]
 [image units (0,0), (0,0), (1,6):] (episodic mem. representation)
 |\$restaurant \$fancy John lobster MaMaison good big|
 see figure 3.5)

[generating paraphrase:]
 |John went - - - MaMaison| (story generator output)
 JOHN WENT TO MAMAISON . (sequence of output words)
 |waiter seated - - John -|
 THE WAITER SEATED JOHN .
 |John asked waiter - lobster -|
 JOHN ASKED THE WAITER FOR LOBSTER .
 |John ate - good lobster -|
 JOHN ATE A GOOD LOBSTER .
 |John paid waiter - - -|
 JOHN PAID THE WAITER .
 |John left waiter big tip -|
 JOHN LEFT A BIG TIP .
 |John left - - - MaMaison|
 JOHN LEFT MAMAISON .

[parsing question:]
 WHAT DID JOHN EAT AT MAMAISON ?
 |John ate - - what MaMaison| (sentence parser output)
 see figure 3.6)

[into cue:] (cue former output)
 |\$restaurant \$fancy John ? MaMaison good ?|
 see figure 3.7)

[retrieving from episodic memory:]
 [image units (0,0), (0,0), (1,6):]
 |\$restaurant \$fancy John lobster MaMaison good big|
 (episodic mem. output)

[generating answer:]
 |John ate - good lobster -| (answer producer output)
 JOHN ATE A GOOD LOBSTER . (see figure 3.8)

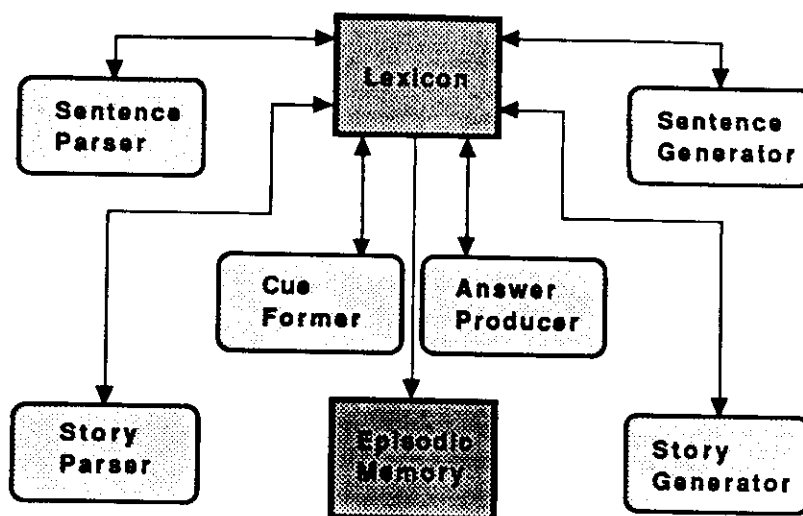


Figure 3.9: **Training configuration.** Each module is trained separately and simultaneously with compatible I/O data. For each training example, an external training supervisor splits up the parsing, paraphrasing, and question answering tasks and forms the input and target patterns for each module, using the current representations in the lexicon. There is no propagation between modules, but they simultaneously perform different parts of the same story processing tasks. All processing modules also modify the same representation set in the lexicon.

3.3 Training and performance

Figure 3.1 depicts DISCERN after it has been trained and it is performing the script processing task. The arrows connecting the modules indicate pathways through which the word, case-role and story representations propagate. During performance, the only weight changes occur in the episodic memory, when new story traces are created.

During training, the system evolves in three ways: the weights in the processing modules change to improve the module's performance in its task, the concept representations in the lexicon adapt to the requirements of the whole system, and the lexical, semantic, and episodic memories self-organize to reflect the taxonomy of the data.

A good use of the modular structure can be made in training DISCERN. Each module can be trained separately and in parallel (figure 3.9). The training data consists of complete examples of story processing. The correct input/output words in parsing, question answering and paraphrasing each story are specified for each module. An external training supervisor forms the input and target patterns for each module, using the current representations in the lexicon. There is no propagation between modules during training, but they are trained with compatible input/output data. If they successfully learn their tasks, they will know how to process each other's output when they are connected for performance.

DISCERN learns to perform script processing quite accurately. After the modules are connected for performance, typically 97-99% of its output words are correct, and the errors

it makes are plausible in terms of human performance (chapter 10).

Part II

PROCESSING MECHANISMS

The processing in DISCERN is performed by FGREP modules. These are simple recurrent or feedforward backpropagation networks with the special property that they automatically develop their input representations while they are learning the processing task. The different versions of the FGREP mechanism are presented and analyzed in the sentence parser subtask, i.e. assigning roles to sentence constituents. Separate sentence data is used for this analysis, different from the script processing data, to better illustrate the properties of FGREP. The performance of the processing modules of DISCERN are then analyzed with the actual script data. Part II concentrates solely on processing mechanisms, bypassing episodic memory and lexical memory. Only the semantic memory part of the lexicon is discussed, and the term "word" and labels such as ball refer to the concepts, not the lexical symbols themselves.

Chapter 4

Developing representations in FGREP-modules

4.1 Backpropagation networks

Backward error propagation is the most extensively studied learning rule for multilayer perceptron systems. A basic backward error propagation network consists of an input layer, an output layer and one or more hidden layers connected in a feed-forward fashion. An input activity pattern is loaded into the input layer and the activity is propagated through the network to the output layer. The learning consists of modifying the connection weights so that correct output patterns are produced for each input pattern [Rumelhart et al., 1986b].

In the forward phase, each unit in the hidden and output layers forms a sum of its input values, weighted by the connection weights, and produces an output value as a semilinear (differentiable and nondecreasing) function of the sum (figure 2). A sigmoid is convenient since it limits the activity values to a certain range and has convenient mathematical properties:

$$o_{ki} = \sigma\left(\sum_l w_{(k-1)li} o_{(k-1)l} + b_{ki}\right) = \frac{1}{1 + e^{-(\sum_l w_{(k-1)li} o_{(k-1)l} + b_{ki})}} \quad (4.1)$$

where o_{ki} is the output of the i :th unit at layer k , $w_{(k-1)li}$ is the weight between the l :th unit in the $(k-1)$:th layer and the i :th unit in the k :th layer and b_{ki} is the bias of the sigmoid (its displacement from the origin) at unit ki . b_{ki} can also be interpreted as a weight on a connection from a unit which is always active at 1.0.

Once the activity pattern is produced in the output layer it is compared to the correct activity pattern, the teaching input. An error signal is formed for each output unit (layer n):

$$\delta_{nj} = f'_{nj}(net_{nj}) (t_j - o_{nj}), \quad (4.2)$$

where t_j is the target activation value for the output unit j and $f'_{nj}(net_{nj})$ denotes the derivative of the semilinear activation function of the output unit j with respect to its total input $net_{nj} = \sum_l w_{(n-1)lj} o_{(n-1)l} + b_{nj}$. If this function is a sigmoid the equation becomes simply

$$\delta_{nj} = o_{nj}(1 - o_{nj})(t_j - o_{nj}). \quad (4.3)$$

Similarly, the error signal is formed for each unit in the previous layers k as

$$\delta_{ki} = f'_{ki}(net_{ki}) \sum_j \delta_{(k+1)j} w_{kij}, \quad (4.4)$$

or in the case of sigmoid activation,

$$\delta_{ki} = o_{ki}(1 - o_{ki}) \sum_j \delta_{(k+1)j} w_{kij}. \quad (4.5)$$

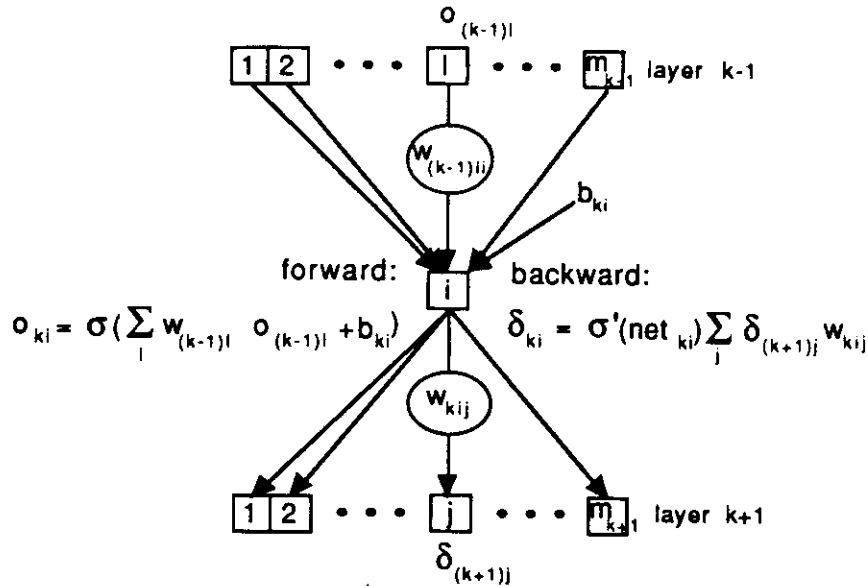


Figure 4.1: Forward and backward propagation

In other words, the error signals are propagated backward through the network. Forward and backward propagation for a general unit i at layer k are illustrated in figure 4.1.

Learning consists of changing each connection weight according to

$$\Delta w_{kij} = \eta \delta_{(k+1)j} o_{ki}, \quad (4.6)$$

where η is the learning rate. If the total error for an input-output pair p is defined as

$$E_p = \frac{1}{2} \sum_j (t_j - o_{nj})^2 \quad (4.7)$$

it can be shown [Rumelhart et al., 1986b] that the Δw_{kij} 's indicate the steepest descent of the total error for this pattern:

$$-\frac{\partial E_p}{\partial w_{kij}} = \delta_{(k+1)j} o_{ki}. \quad (4.8)$$

In intuitive terms, backward error propagation thus consists of determining the amount of error at each output unit, distributing the responsibility of the error throughout the network and changing the weights toward maximally reducing the total error. Since the objective is to learn to associate several input-output pairs, the gradient should be computed over the whole set of inputs before making the weight change (batch update). Moreover, to implement true gradient descent, the weight change should be infinitely small. In practice, if the learning rate parameter η is small enough, the weights can be changed gradually while cycling through the set of input patterns several times (online update). A complete cycle over the training set is the basic unit of training, and referred to as an *epoch*.

Large learning rates induce larger weight changes and faster learning, but often lead to oscillation. A common method for reducing oscillations is to include a momentum term in the weight change:

$$\Delta w_{kij}(t) = \eta \delta_{(k+1)j} o_{ki} + \alpha \Delta w_{kij}(t-1), \quad (4.9)$$

The weight change now carries momentum from previous epochs. Each weight change step reflects the general direction of the error surface, with high-frequency oscillations filtered out.

Because backpropagation networks are based on multiple layers of nonlinear units, they are not limited to linearly separable classifications like the early Perceptron [Rosenblatt, 1962; Minsky and Papert, 1988]. It has been shown that a three-layer network (two layers of weights) can compute any classification [Rumelhart et al., 1986b]. However, the learning algorithm implements gradient descent, which in principle is not a good optimization method. If there are local minima in the error function, the method runs a risk of getting stuck. Surprisingly, this is rarely a problem. Perhaps the error functions in interesting problems with reasonable initial conditions do not have serious local minima, or the non-infinitesimal move along the gradient can escape some of them. Also, using the online update of weights instead of the batch method might find its way around the minima. The robustness of backpropagation against local minima is not very well understood, but fortunately it is not a serious problem in practice. Slow convergence is much more serious, and has received more attention in backpropagation research [Rumelhart et al., 1986b; Hinton, 1987; Fahlman, 1988].

Backpropagation networks have been applied to various tasks with very good results [Rumelhart et al., 1986b; Sejnowski and Rosenberg, 1987; Fahlman, 1988]. The learning algorithm is very robust, and allows many extensions without breaking [Jordan, 1986; Elman, 1990; Almeida, 1987; Pineda, 1987; Pollack, 1987; Williams and Zipser, 1989; Ash, 1989; Mozer, 1988; Plutowski, 1988]. Many tasks can be reduced into a simple mapping, and be made amenable to backpropagation with suitable pre- and postprocessing of data. A major advantage is that the process does not require thorough knowledge of the domain. In many tasks the structure of the domain is not known in enough detail, or the task requires bringing together vast amounts of information from various sources, which makes it hard to develop exact algorithms. However, with backpropagation it is possible to achieve a reasonable level of performance even in these situations. Usually the network can extract a reasonable approximation of the desired input \rightarrow output mapping from a collection of input/output examples.

It would be useful to know what knowledge the network acquires in the process. To this date there is no robust method of extracting the knowledge from the network. This is a serious problem. For example, it may be possible to train a network to do a great job in evaluating loan applications, but we cannot use it because it does not give us exact reasons why it denied a given application.

Another problem concerns the nature of pre- and postprocessing. How should the distributed input and output representations to the network be encoded? Different approaches are evaluated in section 12.6.1. In this chapter we will present a method where global representations are developed automatically by the network itself while it is learning the

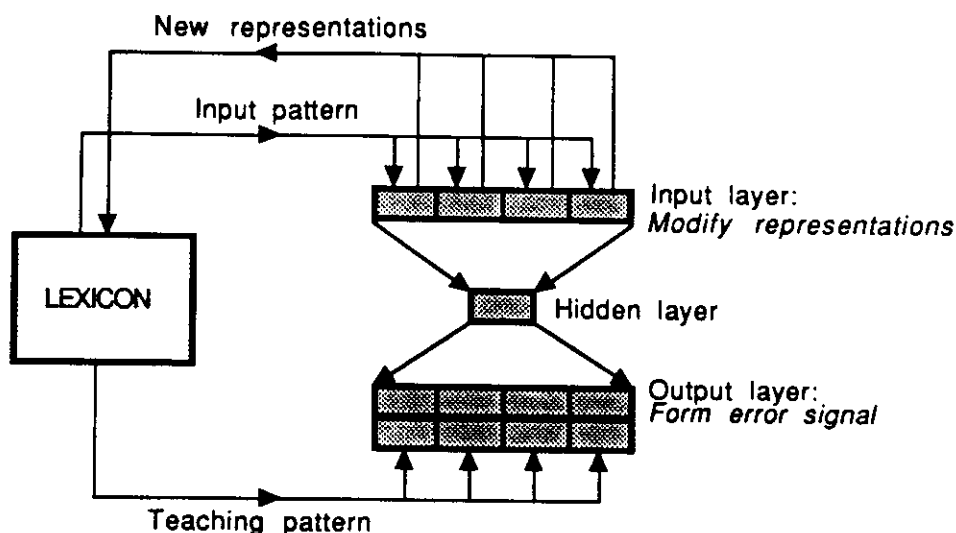


Figure 4.2: **Basic FGREP architecture.** The system consists of a three-layer backpropagation network and an external, global lexicon containing the input/output representations. Each input and output assembly (grey rectangle in the figure) holds a single word representation, taken from the lexicon. In the figure, four words are concatenated in the input and output layer. At the end of each backpropagation cycle, the current input representations are modified at the input layer according to the error signal. The new representations are loaded back to the lexicon, replacing the old ones.

processing task.

4.2 FGREP: forming global representations with extended backpropagation

4.2.1 Basic FGREP architecture

The FGREP mechanism is based on a basic three-layer backward error propagation network (figure 4.2). The network learns the processing task by adapting the connection weights according to the standard backpropagation equations (section 4.1). At the same time, representations for the input data are developed at the input layer according to the error signal extended to the input layer. Input and output layers are divided into assemblies and several items are represented and modified simultaneously.

The representations are stored in an external lexicon network. A routing network forms each input pattern by concatenating the current lexicon entries of the input items; likewise, it forms the corresponding teaching pattern by concatenating the lexicon entries of the target items. Thus each item is represented by the same pattern in the input and in the output of the backpropagation network.

The representation formation process begins with a random lexicon containing no pre-encoded information. During the course of learning, the representations adapt to reflect the implicit regularities of the task. It turns out that single components in the resulting

representations do not necessarily have a clear interpretation. The representation does not implement a classification of the item along identifiable features (as in semantic feature encoding). In the most general case, the representations are simply profiles of continuous activity values over a set of processing units. This representation pattern as a whole is meaningful and can be claimed to code the meaning of that item. The representations for items which are used in similar ways become similar.

4.2.2 Extending backpropagation to the representations

Standard backpropagation produces an error signal for each hidden layer unit. By propagating the signal one layer further to the input layer, the representations can be changed as if they were an extra layer of weights.

In a sense, the representation of an item serves as input activation to the input layer. The activation of an input unit is identical to the corresponding component in the representation. In this analogy, the activation function of an input unit is the identity function and its derivative is one. The error signal can be computed for each input unit as a simple case of the general error signal equation refeq:generrsig :

$$\delta_{1i} = \sum_j \delta_{2j} w_{1ij}. \quad (4.10)$$

where δ_{xy} stands for the error signal for unit y in layer x , and w_{1ij} is the weight between unit i in the input layer and unit j in the first hidden layer. The representations are now changed according to the error signal:

$$\Delta r_{ci} = \eta \delta_{1i} r_{ci}, \quad (4.11)$$

where r_{ci} is the representation component i of item c , δ_{1i} is the error signal of the corresponding input layer unit and η is the learning rate. This is the standard adjustment in backpropagation, only instead of weights we are now adjusting representations. In other words, the *representation learning is implemented as an extension of the back propagation algorithm*. While the weight values are unlimited, the representation values must be limited between the maximum and minimum activation values of the units. The new value for the representation component i of item c is obtained as

$$r_{ci}(t+1) = \max[o_l, \min[o_u, r_{ci}(t) + \Delta r_{ci}]], \quad (4.12)$$

where o_l is the lower limit and o_u is the upper limit for unit activation.

Note that the backpropagation algorithm “sees” the representations simply as an extra layer of weights. Since the representations adapt according to the error signal, there is reason to believe that *the resulting representations will effectively code properties of the input elements which are most crucial to the task*.

4.2.3 Reactive training environment

The process differs from ordinary backpropagation in that *both the input and the teaching patterns are changing*. An input pattern is formed by drawing the current representations

of the input items from the lexicon and loading them into the input assemblies (figure 4.2). The activity is propagated through the network to the output layer, where the error signal is formed by comparing the output pattern to the teaching pattern, which is also formed by selecting representations from the lexicon. The error signal is propagated back to the input layer, changing both weights and the input item representations along the way. Next time the *same input* occurs, the output will be closer to the *same teaching pattern*. The modified input representations are now put back into the lexicon, replacing the old ones and thereby *changing the next teaching pattern for the same input*. The shape of the error surface is changing at each step, i.e. backpropagation is shooting at a moving target in a reactive training environment.

It turns out that as long as the changes made in the process are small, the process converges nevertheless. The learning time appears to be about the same as in the ordinary case. The modifiable input patterns form an additional set of parameters which the system can use to learn the task. The changes in the error surface are a form of noise (a noisy error signal), which backpropagation in general tolerates very well.

It is conceivable that the representations might converge into a trivial solution, where they are all identical. Such a situation has never occurred in our experiments, and our sense is that it is very unlikely to occur when the representations are initially random. It seems that starting from a random point in the representation set space, the process is much more likely to get into a nontrivial stable solution. Apparently these solutions are more numerous and more evenly distributed in the solution space than the trivial solutions. Moreover, if some of the representations are fixed (e.g. the “blank” representation at all-zero, and the “don’t care” representation at all 0.5), identical representation sets are excluded altogether.

4.3 Subtask: Assigning case roles to sentence constituents

DISCERN uses case-role representation of sentences as an intermediate representation, facilitating modular overall architecture. In this chapter, different versions of FGREP are analyzed in the context of the case-role assignment task: basic FGREP (section 4.4), cloning synonymous word instances (section 4.5), and dealing with sequential natural language input/output (section 4.6). The case-role assignment task is introduced here in its basic form, and modified later for sequential input.

Case-role representation of sentences is based on the theory of thematic case roles [Fillmore, 1968], adapted for computer modeling in conceptual dependency theory [Schank and Abelson, 1977; Schank and Riesbeck, 1981]. In the basic version of the case-role assignment task the syntactic structure of the sentence is given and consists of e.g. the subject, verb, object and a with-clause. The task is to decide which constituents play the roles of agent, patient, instrument and patient modifier in the act (figure 4.3). This task requires forming a shallow semantic interpretation of the sentence.

For example, in *The ball hit the girl with the dog*, the subject *ball* is the instrument of the *hit-act*, the object *girl* is the patient, the with-clause, *dog*, is a modifier of the patient, and the agent of the act is unknown. Role assignment is context dependent: in *The ball moved* the same subject *ball* is taken to be the patient. Assignment also depends on


INPUT:	Syntactic constituents	Subj. ball	Verb hit	Object girl	With dog	
						
OUTPUT:	Case-role assignment	Agent -	Act hit	Patient girl	Instr. ball	Modif. dog

Figure 4.3: Assigning case roles to sentence constituents. The example sentence is The ball hit the girl with the dog.

the semantic properties of the word. In *The man ate the pasta with cheese* the with-clause modifies the patient but in *The man ate the pasta with a fork* the with-clause is the instrument. In yet other cases the assignment must remain ambiguous. In *The boy hit the girl with the ball* there is no way of telling whether ball is an instrument of hit or a modifier of girl.

Case-role assignment is an example of a cognitive task which is well suited for modelling with connectionist systems. The task requires taking into account all the positional, contextual and semantic constraints simultaneously, which is what the connectionist systems are particularly good at.

In [McClelland and Kawamoto, 1986] the authors describe a system which learns to assign case roles to sentence constituents. Their architecture differs from FGREP in several ways (discussed in section 12.2.1), but most significantly, the distributed representations for the words in their system were pre-encoded using semantic feature classification. In this approach, each word is classified along a number of dimensions (such as human, softness, gender etc.), and the result bits are concatenated into one long representation vector (see section 12.6.1).

The same task with the same data was used to test FGREP, partly because it provides a convenient comparison to a system using semantic feature encoding. The task was restricted to a small repertoire of sentences studied in the original experiment. The sentence generators are depicted in table 4.1 and the noun categories in table 4.2 (the code and category data used for generating the sentences are listed in appendix E.1.3).

The sentence frames and the noun categories are not visible to the system: they are only manifest in the combinations of words that occur in the input sentences. To do the case-role assignment properly the system has to figure out the underlying relations and code them into the representations.

In this particular task, the teaching input is made up from the input sentence constituents (figure 4.4). This structure is by no means necessary for learning the representations. The required output of the network could be anything and the FGREP method would work the same. A discriminatory or "pigeonholing" task, such as case-role assignment, is actually harder than a general I/O mapping task because of the reactive training effect.

Gen.	Sentence Frame	Correct case roles
1.	The human ate.	agent
2.	The human ate the food.	agent-patient
3.	The human ate the food with the food.	agent-patient-modif
4.	The human ate the food with the utensil.	agent-patient-instr
5.	The animal ate.	agent
6.	The predator ate the prey.	agent-patient
7.	The human broke the fragileobj.	agent-patient
8.	The human broke the fragileobj with the breaker.	agent-patient-instr
9.	The breaker broke the fragileobj.	instr-patient
10.	The animal broke the fragileobj.	agent-patient
11.	The fragileobj broke.	patient
12.	The human hit the thing.	agent-patient
13.	The human hit the human with the possession.	agent-patient-modif
14.	The human hit the thing with the hitter.	agent-patient-instr
15.	The hitter hit the thing.	instr-patient
16.	The human moved.	agent-patient
17.	The human moved the object.	agent-patient
18.	The animal moved.	agent-patient
19.	The object moved.	patient

Table 4.1: Sentence generators. The generators are shown as sentence frames, with one to three noun slots. Each slot can be filled with any of the nouns in the specified category (table 4.2), and each slot has a predetermined case role. For instance, The human ate the food generates 4×4 different sentences, all with the case-role assignment human = agent, food = patient.

Category	Nouns
human	man woman boy girl
animal	bat chicken dog sheep wolf lion
predator	wolf lion
prey	chicken sheep
food	chicken cheese pasta carrot
utensil	fork spoon
fragileobj	plate window vase
hitter	bat ball hatchet hammer vase paperwt rock
breaker	bat ball hatchet hammer paperwt rock
possession	bat ball hatchet hammer vase dog doll
object	bat ball hatchet hammer paperwt rock vase plate window fork spoon pasta cheese chicken carrot
thing	desk doll curtain human animal object

Table 4.2: Noun categories.

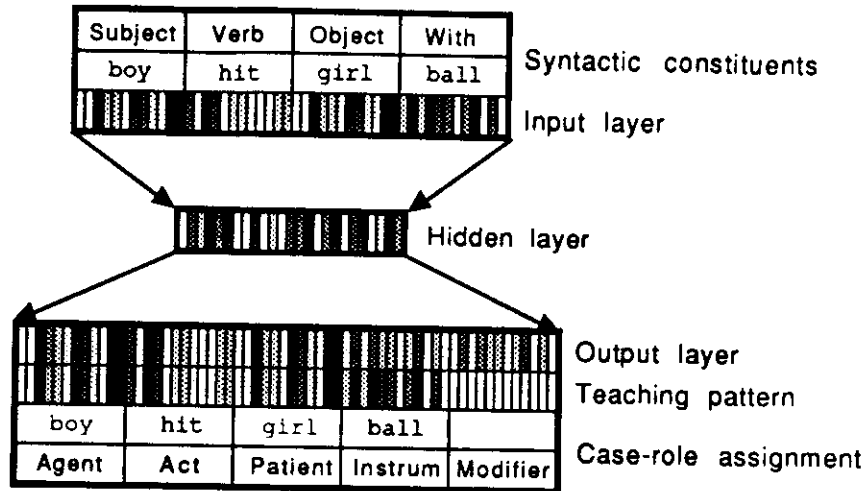


Figure 4.4: Snapshot of basic FGREP simulation. The input and output layers of the network are divided into assemblies, each of which holds one word representation at a time. Each unit in an input assembly is set to the activity value of the corresponding component in the lexicon entry. The input layer is fully connected to the hidden layer and the hidden layer to the output layer. Connection weights are omitted from the figure. If the network has successfully learned the task, each output assembly forms an activity pattern which is identical to the lexicon representation of the word that fills that role. The correct role assignment is shown at the bottom of the display. This pattern forms the teaching input to the network. Grey-scale values from white to black are used in the figure to code the unit activities in the range [0,1].

4.4 Properties of FGREP-representations

4.4.1 Simulations

The learning is fairly insensitive to the simulation parameters and the system configuration parameters. The online version of backpropagation without momentum turns out to be the most efficient training method. Best results are obtained by presenting the sentences within each epoch in different random order, and by gradually reducing the learning rate. The results reported in the following three sections are from the run with the learning rate $\eta = 0.1$ for the first 200 epochs, 0.05 until 500, and 0.025 until 600.

The number of units in the representation and the number of hidden units are not crucial either: as few as 5 and as many as 100 were tried. If more hidden units are used, the task performance and damage resistance improve slightly and the learning in general is faster. Decreasing the number of hidden units on the other hand places more pressure on the representations, and they become more descriptive faster. In general, the best results are obtained when the number of hidden units is about half the number of units in the input layer. In the example simulation, 12 units were used for each representation (i.e. 48 input units total) and 25 for the hidden layer. The representation components were initially uniformly distributed in the interval [0,1] and the connection weights within [-1,1]. Figure 4.4 shows a snapshot of the simulation after a real-time display on an HP 9000/350 workstation. Key parts of the training code are listed in appendix E.1.1, the simulation specification files and

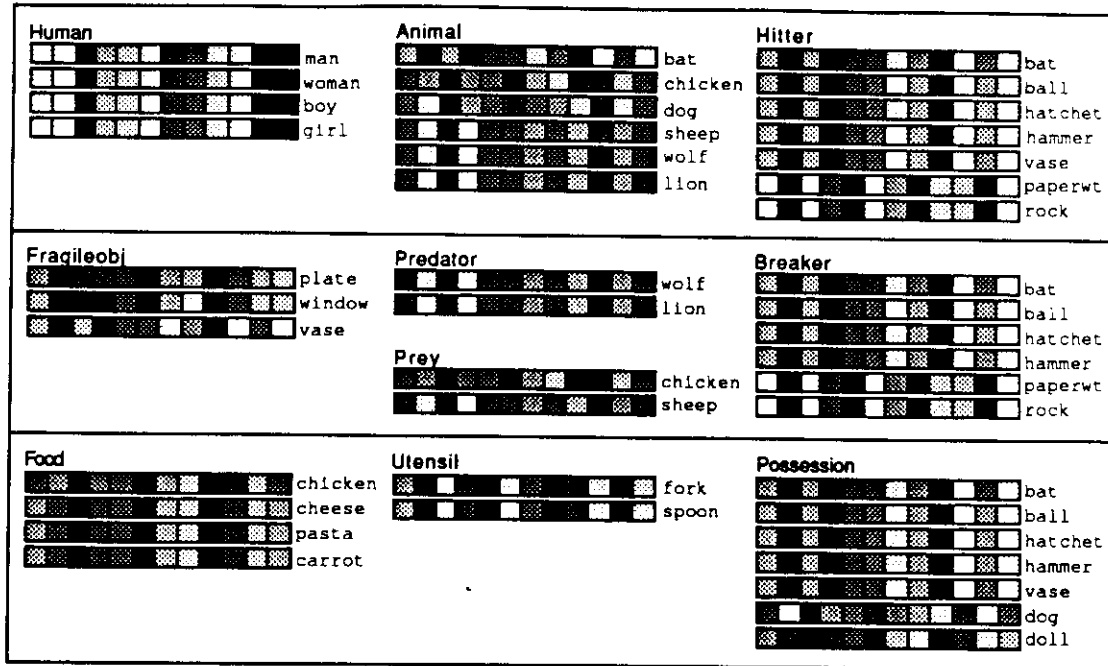


Figure 4.5: Final representations. The representations for the synonymous words {man, woman, boy, girl}, {fork, spoon}, {wolf, lion}, {plate, window}, {ball, hatchet, hammer}, {paperwt, rock} and {cheese, pasta, carrot} have become almost identical.

samples of the data files are listed in appendix E.1.4.

The next four sections concentrate on a baseline simulation, where 1439 of the 1475 sentences produced by the sentence generators were used for training and 38 were reserved for testing. Generalization as a function of the training set size is discussed in section 4.4.6.

4.4.2 Final representations

Starting from random representations, the similarity of the nouns belonging to the same category first increases rapidly until the changes begin to cancel out. The categorization is in a dynamic equilibrium (i.e. representations change but categorization does not improve) while the task performance improves. The decreasing error signal and learning rate eventually allow fine tuning the representations and they converge into a stable, descriptive categorization. Figure 4.5 displays the final noun representations, organized according to the categories. With different initial configurations, the final set of representations would look different, but the overall similarities of the representations and the performance of the system would be the same.

Some words belong exactly to the same categories and consequently occur exactly in the same contexts. They are indistinguishable in the data and their representations become identical. {man, woman, boy, girl} forms one such group, {fork, spoon}, {wolf, lion}, {plate, window}, {ball, hatchet, hammer}, {paperwt, rock} and {cheese, pasta, carrot} others. If there is at least one difference in the usage of two nouns, their representations

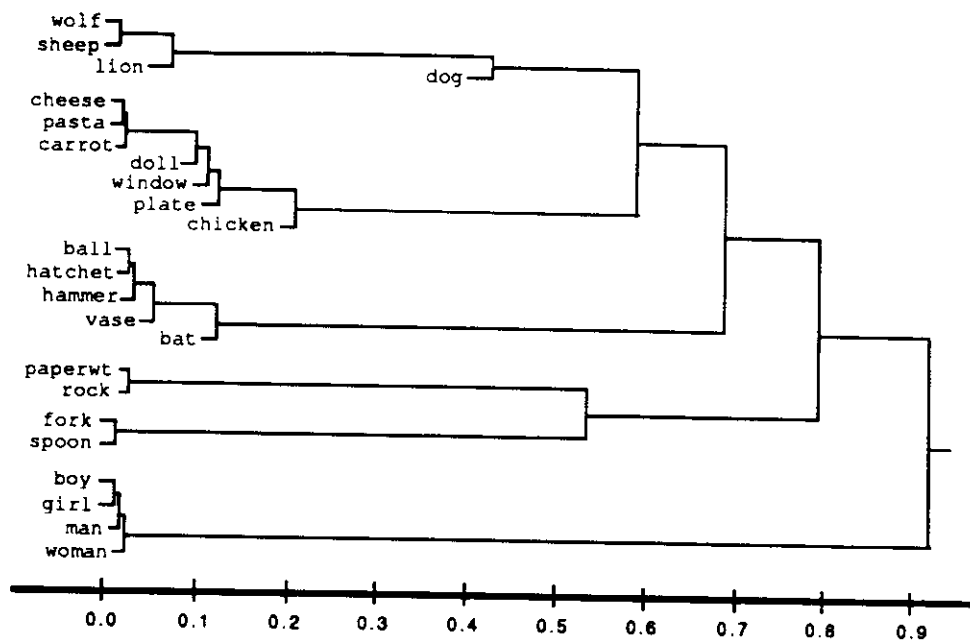


Figure 4.6: Merge clustering the representations. Step by step, the clusters with the shortest single linkage distance were merged. Distance is indicated on the horizontal scale. There are five prominent clusters with very small distances between items: animals, humans, utensils, two different types of hitters and a combination of foods and fragile-objects. Ambiguous words (*chicken*, *bat*) and words with an unusual use (e.g. *dog*, which is a possession but not a hitter) do not fit very well into any category, and they are merged later in the process. Eventually the clusters are fused together, but only by bringing together very different items.

become different. The discriminating input modifies one of the representations while the other one remains the same.

Since each noun belongs to several categories, its representation can be seen as evolving from the competition between the categories. This is clearest on the part of the ambiguous nouns *chicken* and *bat*, which on the one hand are both animals, but *chicken* is also food and *bat* is a hitter. The representation is a combination of both, weighted by the number of occurrences of each meaning (there are more plausible ways to deal with ambiguous words; see section 11.5). On the other hand, the fact that there is a common element in two categories tends to make all representations of the two categories more similar. The properties of one word are generalized, to a degree, to the whole class.

Note that the categorization of a word in figure 4.5 is formed outside the system and is independent of the task, other categories and other words. The system itself is not attempting categorization, it is forming the most efficient representation of each word for a particular task. Interestingly, hierarchical merge clustering algorithm [Hartigan, 1975] (code in appendix E.2) finds optimal clusters quite similar to the noun categories (figure 4.6).

Inspection of the representations in figure 4.5 suggests that a single unit does not play a crucial role in the classification of items. The fact that a word belongs to a certain category is indicated by the *activity profile as a whole*, instead of particular units being on or off. The

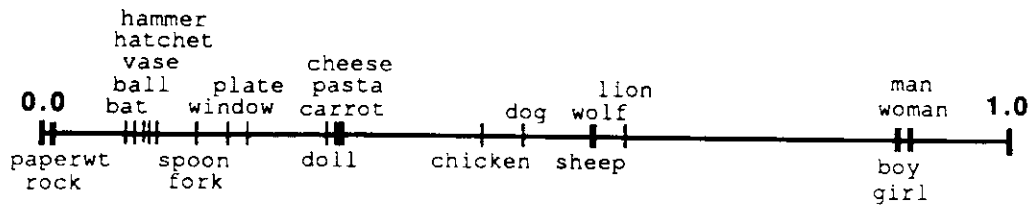


Figure 4.7: **Categorization by unit 11.** The words are placed on a continuous line $[0,1]$ according to the value of the last unit in their representations (the units are numbered 0 - 11).

representations are also extremely *holographic*. The whole categorization is visible even in the values of a single unit (figure 4.7).

Each unit provides a unique perspective into the words, by coding slightly different properties of the word space. Combining the values of two units thus provides an even more descriptive categorization (figure 4.8), and the complete representation can be seen as a combination of twelve slightly different viewpoints into the word space.

To obtain insight into this combined categorization, we first have to map the 12 - dimensional representation vectors into two dimensions. One way to do this is Kohonen's self-organizing feature mapping [Kohonen, 1984]. This method is known to map clusters in the input space to clusters in the output space. The map is topological, i.e. the distances in the map are not comparable (more dense regions are magnified), but the topological relations of the input space are preserved (feature maps are discussed in more detail in chapter 6).

The feature map shows the same clusters that were used in generating the input (figure 4.9). The ambiguous nouns and nouns belonging to several categories are mapped between the categories. *chicken* is mapped between *animal* and *food*, *bat* between *animal* and *hitter*, and also *vase* between *fragile-obj* and *hitter*.

It is very hard to name the properties that the individual units are coding. In [Hinton, 1986] the author was able to give semantic interpretation for some of the units in the hidden layer representation, although he points out that the system develops its own microfeatures, which may or may not correspond to ones that humans would use to characterize data. Similar results have been reported in [Pollack, 1988; Sejnowski and Rosenberg, 1986].

Our results suggest that with complex data, the microfeatures in the resulting representation are identifiable only accidentally. The network is trying to code several dimensions of variation (i.e. feature dimensions) into a lower-dimensional representation, and it has to distribute the features over several units. As a result, each individual unit becomes sensitive to a *combination of several features* which is very unlikely to exactly match an established term, although partial matches are possible. For example, unit 11 seems to separate objects which are "animate", but the separation is not clear-cut and has a lot of finer structure (figure 4.7). The network does not lose the information about animateness if this unit becomes defective (section 4.4.4), so it is incorrect to say that the unit codes the "animate-feature" of the input data.

The holographic property does not mean that the representations are uninterpretable. Instead of labeling single units, the interpretation requires more sophisticated techniques

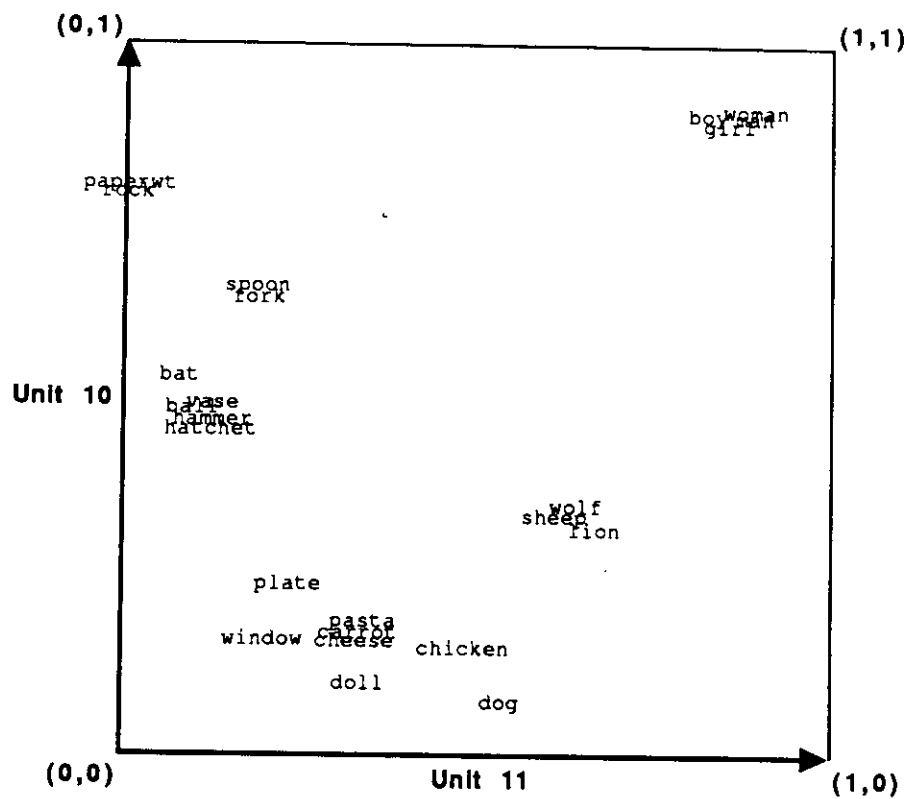


Figure 4.8: **Categorization by units 10 and 11.** The words are mapped on the unit square according to the values of the last two units in their representations.

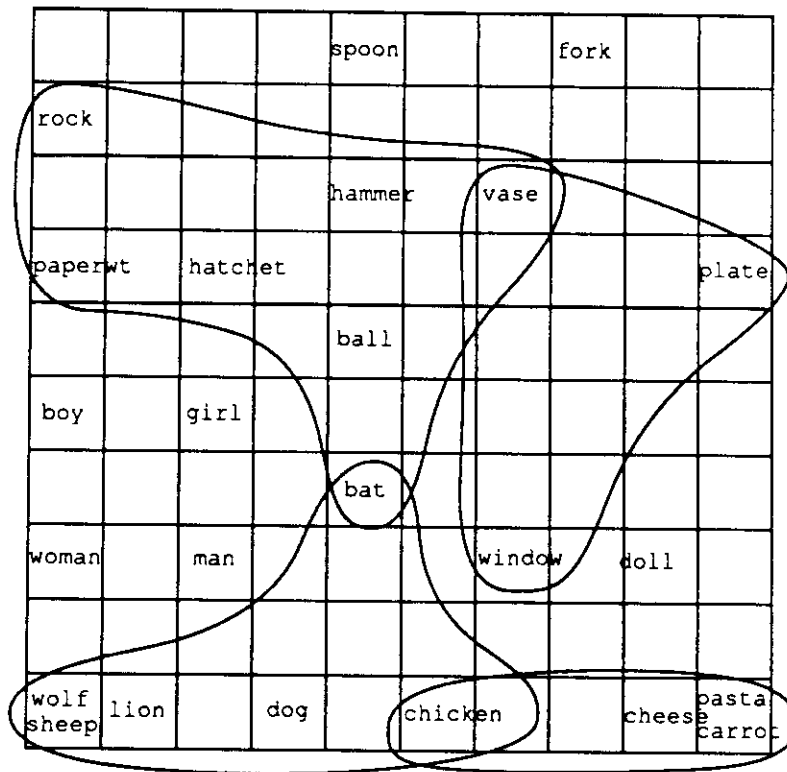


Figure 4.9: **2-D Kohonen-map of the representations.** Labels indicate the maximally responding unit in the 10×10 feature map network for each representation vector. The map was formed in 15,000 epochs, where the neighborhood radius was decreased from 4 to 1 and the learning rate from 0.5 to 0.05 in the first 1,000 epochs, and to zero during the remaining epochs. The code for forming the map is included in appendix C.2.1, with simulation specifications in E.3.1.

Gen.	Input			$E_i < 0.15$	E_{avg}
1.	man	ate		88	.067
1.	girl	ate		88	.066
2.	woman	ate	cheese	100	.018
2.	woman	ate	pasta	100	.018
3.	woman	ate	chicken	100	.015
3.	man	ate	pasta	100	.021
4.	girl	ate	pasta	100	.023
4.	boy	ate	chicken	100	.025
5.	dog	ate		83	.068
5.	sheep	ate		82	.065
6.	lion	ate	chicken	97	.037
6.	lion	ate	sheep	98	.034
7.	woman	broke	window	100	.014
7.	boy	broke	plate	100	.014
8.	man	broke	window	100	.016
8.	boy	broke	plate	100	.014
9.	paperwt	broke	vase	100	.023
? 9.	bat	broke	plate	100	.028
? 10.	bat	broke	window	68	.182
10.	wolf	broke	plate	100	.023
11.	vase	broke		93	.039
11.	window	broke		100	.024
12.	man	hit	pasta	100	.009
12.	girl	hit	boy	100	.023
? 13.	man	hit	girl	77	.093
? 13.	man	hit	woman	77	.092
14.	woman	hit	bat	100	.008
14.	girl	hit	vase	100	.011
15.	hatchet	hit	pasta	100	.008
15.	hammer	hit	vase	100	.014
16.	man	moved		93	.054
16.	woman	moved		93	.054
17.	woman	moved	plate	100	.010
17.	girl	moved	pasta	100	.010
? 18.	bat	moved		80	.118
18.	dog	moved		87	.047
19.	doll	moved		100	.025
19.	desk	moved		100	.020
Average:				95	.038

Table 4.3: Performance, familiar sentences. The leftmost entry in each row identifies the generator which produced the sentence (referring to table 4.1). The first number after each sentence indicates the percentage of output units whose values were within 0.15 of the correct output value (which ranged from 0 to 1). The second number indicates the average error per unit. Ambiguous sentences are marked with "?". The test sentence sets are identical to those used in [McClelland and Kawamoto, 1986].

which look at complete representation vectors, such as cluster analysis and feature mappings, or the use of functional criteria [van Gelder, 1989]. It might also be possible to rotate the representation vector so that some of the representation dimensions (i.e. units) match the feature dimensions discovered by the network, and subsequently label the units of the rotated vector.

4.4.3 Performance in the task

The performance of the system in the role assignment task was tested with two test sets. The first one consisted of two sentences from each generator which had been used in the training, and the second one consisted of the test sentences which the network had not seen before. Tables 4.3 and 4.4 show the results for each sentence. The testing code is included in appendix E.1.2.

The system learns the correct assignment for most sentences. Note that perfect perfor-

Gen.	Input			$E_i < 0.15$	E_{avg}
1.	boy	ate		88	.066
1.	woman	ate		88	.066
2.	woman	ate	chicken	100	.018
2.	man	ate	chicken	100	.018
3.	woman	ate	chicken	100	.015
3.	boy	ate	carrot	100	.012
4.	man	ate	chicken	100	.025
4.	woman	ate	carrot	100	.023
5.	bat	ate		67	.186
5.	chicken	ate		68	.116
6.	wolf	ate	chicken	98	.036
6.	wolf	ate	sheep	98	.034
7.	girl	broke	plate	100	.014
7.	woman	broke	plate	100	.014
8.	man	broke	vase	100	.016
8.	girl	broke	vase	100	.016
9.	hammer	broke	vase	100	.018
9.	ball	broke	vase	100	.018
?10.	bat	broke	vase	68	.192
10.	dog	broke	plate	98	.032
11.	plate	broke		100	.026
11.	plate	broke		100	.026
12.	boy	hit	girl	100	.023
12.	girl	hit	carrot	100	.009
?13.	man	hit	boy	77	.092
13.	boy	hit	woman	100	.026
14.	girl	hit	curtain	100	.011
14.	girl	hit	spoon	100	.007
15.	paperwt	hit	chicken	100	.014
15.	rock	hit	plate	100	.014
16.	boy	moved		93	.054
16.	girl	moved		93	.055
17.	man	moved	window	100	.011
17.	girl	moved	hammer	100	.011
18.	wolf	moved		82	.062
18.	sheep	moved		82	.062
19.	paperwt	moved		88	.056
19.	hatchet	moved		98	.024
Average:				94	.040

Table 4.4: Performance, unfamiliar sentences.

mance is not possible with this data, because some of the sentences are ambiguous. In these cases the system develops an intermediate output between the two possible interpretations, indicating a degree of confidence in the choices. One such case is presented in the snapshot of figure 4.4. *Ball* can be either the instrument of *hit* or a possession of *girl*. In most similar occasions it is the instrument, and the network develops a slightly stronger representation in the instrument assembly. The system also performs poorly with the animal meaning of *bat*. Because a vast majority of the occurrences of *bat* are *hitters*, its pattern becomes more representative of *hitter* than *animal*.

Direct performance comparison with [McClelland and Kawamoto, 1986] is tricky because of different architectures and goals. McClelland and Kawamoto's system was based on binary units, where 96% of the output units should be off on the average. 99% of all the output units were correct in their experiment, but only 85% of the units that should have been on were on. On the other hand, their system performed semantic enrichment of the representations at the output, as well as disambiguation of the different conceptual senses of *bat* and *chicken*, which had different representations in their data. These tasks were not studied in this FGREP experiment.

4.4.4 Damage resistance

The robustness of the representations against damage was tested by eliminating the last n units from each input assembly. These units were fixed at 0.5, the "don't care" value. Figure 4.10 shows the decline in performance as more and more units are eliminated. The decline is very gradual, and approximately linear. This is partly due to the general robustness of hidden-layer networks but also partly due to the fact that the representation is not coded into feature-specific units, but is distributed over all units in a holographic fashion. Eliminating a unit means removing one classification perspective, and these perspectives are apparently additive.

With a third of the input units removed, the system still gets 77% of the output within 0.15 of the correct value. The output patterns are still mostly recognizable at this level. Note also that even with all input units eliminated, i.e. without any information at the input layer the system still performs above chance level. Information about the input space distribution has been stored in the weights, and the network produces a best guess, i.e. an average of all possible outputs.

4.4.5 Creating expectations about possible contexts

The whole pattern in the input and teaching layers has an effect on how each input item is modified during the backpropagation cycle. The context of an input item can therefore be defined operationally as the whole input-output representation.

The representation for an item is determined by all the contexts where that item has been encountered. Consequently, the lexicon entry for that item is also *a representation of all these contexts*. The more frequent the context, the stronger is its trace in the representation. When a word is input into an FGREP module, a number of expectations about the context are

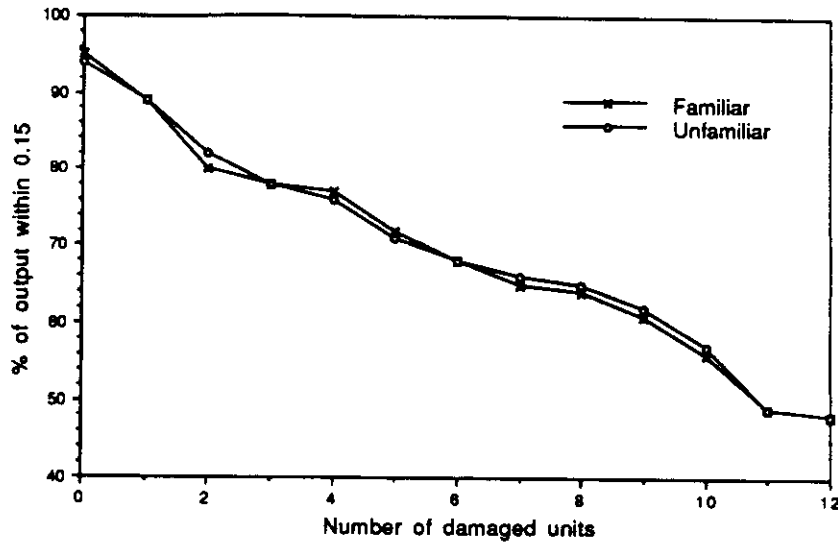


Figure 4.10: **Damage resistance.** The Familiar and Unfamiliar data sets are listed in tables 4.3 and 4.4. The horizontal axis indicates the number of units eliminated from the 12-unit representation, and the vertical axis indicates the percentage of output units which were within 0.15 of the correct value.

automatically created at the output with different degrees of confidence. The expectations of different input words are combined to produce the total output pattern.

Expectations embedded in a single word are displayed when that word is used alone as the input (figure 4.11). The resulting pattern at the output layer indicates the most likely context. As can be seen from the figure, the representations and the network have captured the fact that a likely agent for *ate* is human, patient is food, instrument is utensil, and that food can be eaten with another food.

Being able to create expectations automatically and cumulatively from the input representations turns out to be useful in building larger language understanding systems. Such distributed expectations could replace the symbolic expectations traditionally used in natural language conceptual analyzers, e.g. [Dyer, 1983].

4.4.6 Generalization

The term generalization commonly means processing inputs which the system has not seen before. In most cases this means extending the processing knowledge into new input patterns, which are different from all training patterns. The generalization in FGREP has a different character. The network *never has to extend its processing into very unfamiliar patterns, because the generalization has already been done on the I/O representations.*

If an unfamiliar sentence is meaningful at all, its representation pattern is necessarily close to something the network has already seen. This is because FGREP develops simi-

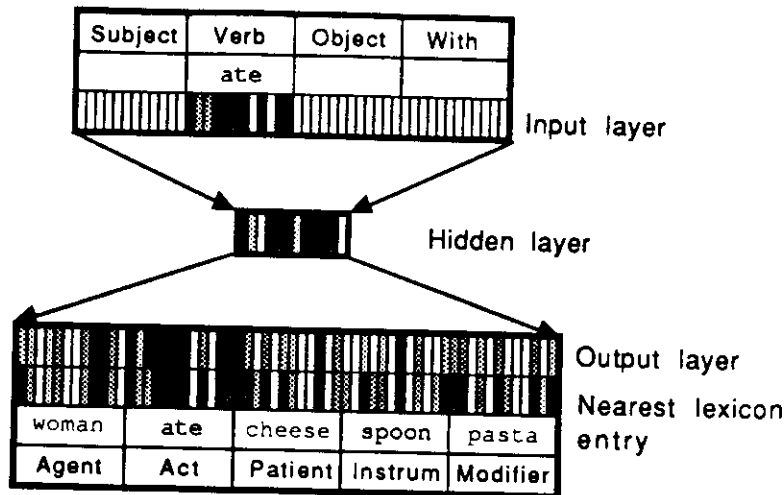


Figure 4.11: Expectations embedded in the word *ate*. The bottom layer shows the lexicon entries which are closest to the output generated by the network (Euclidian distance of normalized vectors). The effect is more pronounced when a small hidden layer is used. In this example a network with 12 hidden units was trained for 50 epochs with a learning rate 0.1.

lar representations for similarly behaving words. For example, the network has never seen *The man ate the chicken with a fork*, but its representation is very close to the familiar sentence *The girl ate the pasta with a spoon*, since the representation for *girl* is equivalent to *man*, *fork* to *spoon*, and *chicken* is very much like *pasta*. In more general terms, the system can process the word *x* in situation *S*, because it knows how to process the word *y* in situation *S*, and the words *x* and *y* are used similarly in a large number of other situations.

If the pattern is far from familiar, the sentence cannot be meaningful in the microworld of the training data. E.g. *The hammer ate the window with the boy* would have a drastically different pattern. Given the experience the network has about hammers, eating, windows and boys, this sentence would be very unlikely to occur, and the network would have difficulty processing it. A sentence like this could not be generated by the sentence generators (table 4.1). It does not belong to the input space the system is trying to learn, i.e. it makes no sense in the microworld.

As a result, the FGREP system processes both the familiar and unfamiliar test sentences *at the same level of performance* (tables 4.3 and 4.4). This is in contrast to systems using representations with precoded, fixed semantic features, such as McClelland and Kawamoto's. Even though two words are functionally equivalent in the input data, in their system the feature representations remain different, and the same level of generalization is much harder to achieve. On the other hand, FGREP loses the ability to distinguish between equivalent words (this problem is discussed in section 4.5).

An interesting question is, how well can FGREP learn the representations and how well does it generalize if it is trained with only a small subset of the input data? This was tested in a series of simulations. Equal number of sentences from each generator were selected randomly for the training, including all sentences in the Familiar set and excluding all in the

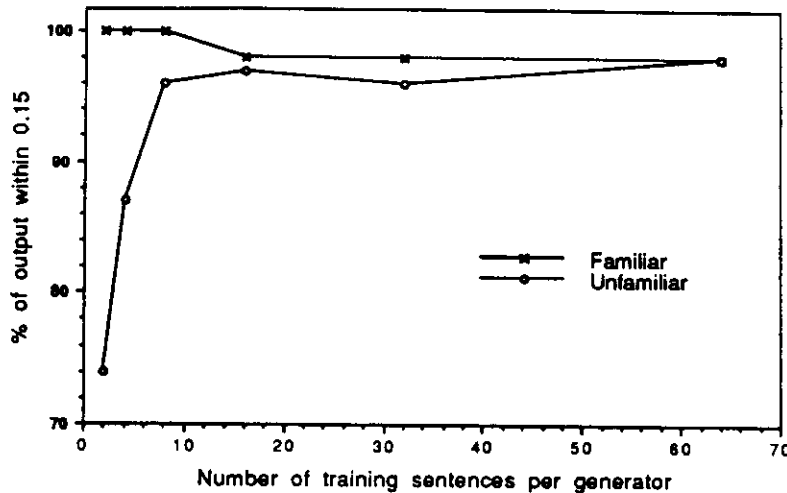


Figure 4.12: Performance as a function of the training set size. The Familiar and Unfamiliar data sets are listed in tables 4.3 and 4.4. The horizontal axis indicates the number of sentences per generator that were used in the training, and the vertical axis indicates the percentage of output units which were within 0.15 of the correct value. The networks were trained for 380,000 input sentence presentations with 0.1 learning rate and another 190,000 presentations with 0.05. This is a total of 375 to 15,000 epochs, depending on size of the training set.

Unfamiliar set. When a generator produced fewer sentences than needed, multiple copies were used.

The performance as a function of the training set size is plotted in figure 4.12. With very small training sets the system learns the idiosyncrasies of the training sentences, and generalizes poorly. Generalization improves very quickly as more training data is included. A critical mass is reached at around eight sentences per generator, which covers approximately 10% of the input space. At this point the system gets 96% of the unfamiliar sentence output within 0.15, and adding more training data does not significantly improve performance.

Even with the very incomplete training data, the final representations end up reflecting the similarities of the words very well. Only the fine tuning of the equivalent words is lost, i.e. their representations are very close but not exactly the same. This is because these words no longer have exactly the same distribution in the training data. The conclusion is that as long as the training data constitutes a good statistical sample of the I/O space, FGREP will develop meaningful representations. The similarities are more and more coarse the smaller the training set, because the asymmetries in the data are coded in. In this particular task and data, *a sample of 10% was large enough to develop meaningful representations, which allowed the system to generalize correctly to the last 90% of the input space.*

The results in figure 4.12 are actually better than those presented in tables 4.3 and 4.4, because the training data was selected differently. In the generalization experiment, the approach was to train the system *with all generators equally*. In the baseline simulation

(section 4.4.1), where the training set was almost complete, the system was effectively trained with the *actual sentence distribution*. Since some generators produce 3 different sentences while others generate 728, different performance figures were obtained for the two cases. A more appropriate test for the baseline system is to test it on the complete set of sentences: the average error is 0.023, while 97% of the input units are within 0.15 (table 4.5). Both types of tests will be used in later sections.

4.5 ID+content: cloning synonymous word instances

4.5.1 Meaning and identity

The FGREP approach is based on the philosophy that *the meaning of a word is manifest in how it is used*. Learning a language is learning the use of the language elements: *language is a skill*. An FGREP representation is defined by all the contexts where the word has been encountered, and it determines how the word behaves in different contexts. The representation evolves continuously as more experience about the word is gained. In other words, FGREP extracts the meaning of the word from its use, and encodes it into the representation.

No two words are used exactly in similar ways in real world situations, and in principle, the meanings of two words are always distinguishable by their use. If there is any difference in the usage, the FGREP process will develop different representations for the words. However, an artificial intelligence system can be exposed only to limited experience, and keeping the representations separate becomes a problem. It is unwieldy to try to generate training sets which would allow enough differences to develop between similar word representations. For example, when the FGREP system reads **The boy hit the girl with the ball** (figure 4.4), it produces an output pattern which is very close to correct, but it is impossible to tell just by looking at the patterns at each output assembly, whether the humans in the actor and patient slots are **man**, **woman**, **boy** or **girl**, and whether the instrument/modifier is **ball**, **hatchet** or **hammer**. Their representations are almost exactly the same.

Word discrimination of the system was measured by finding the closest lexicon representation (in Euclidian distance) for each output assembly and counting how often this was the correct one. The results are shown in table 4.5.

Even though the system has learned the mapping task very well (97% of the output units are within 0.15), it produces a pattern which is closest to the correct output word only about 77% of the time. A closer look reveals that the output words which have unique meanings are correct, but only 59% of the words which have synonyms (i.e. equivalent words such as **man**, **woman**, **boy**, **girl**) are closest to the correct word. The best word discrimination was achieved at around the 100th epoch during training, when the network had learned the case-role assignment task fairly well but the representations had not yet completely converged, providing for enough differences to keep the representations separate. Even in the end the discrimination between synonymous words remains above chance level, because the training data was not completely symmetric, and minute differences remain in their representations. Also, the generalization networks of figure 4.12, which were trained with even more incomplete data, can separate the words much better. We arrive at the

Data	All words	Synonyms	$E_i < 0.15$	E_{avg}
Familiar	75	46	95	.038
Unfamiliar	75	41	94	.040
Complete	77	59	97	.023

Table 4.5: **Performance of basic FGREP.** The Familiar and Unfamiliar data sets are as before, Complete contains all sentences produced by the generators. The first column indicates the percentage of correct words out of all output words, the second shows the percentage for words with synonyms. The third column indicates the percentage of output units which were within 0.15 of the correct value, and the last column shows the average error per output unit.

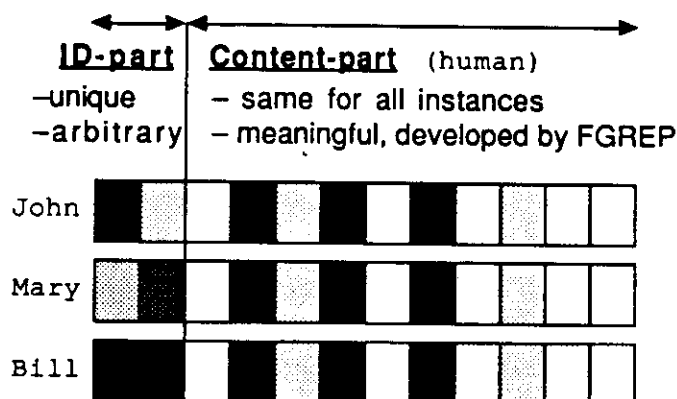


Figure 4.13: **Cloning word instances.** Instances John, Mary and Bill are created from the prototype word human.

curious conclusion that the more extensively the system has experienced the use of the input items, the worse it can keep track of the identities of items which have similar meanings.

The identity problem is even more fundamental in cases where it is necessary to temporarily create several distinct instances of the same item. For example, in *The man who helped the boy blamed the man who hit the boy* it is crucial to know that the man-who-helped is the same as the man-who-blamed, but different from the man-who-hit. Yet there is no way of telling this from the meaning of *man*. It seems necessary to complement the meaning of the item, as extracted from its use, with a surface-level tag which provides a distinct identity for the item.

4.5.2 Composing instances from ID and Content

The simplest way to maintain distinct identities for each word is to designate a subset of representation components for this purpose. The representation now consists of two parts: the content part, which is developed via the FGREP process and which encodes the meaning of the word, and the ID part, which is unique for each instance of the same prototype word (figure 4.13).

Before training, a set of *prototype words* (e.g. *human*) is selected, i.e. the words that we later want to make clones of (e.g. *John*, *Mary*, *Bill*). During training, the units within the ID part of the prototype words are set up randomly for each input presentation, and the network is required to produce the same ID pattern at its output. In effect, the network is trained to process any ID pattern in a prototype word by passing the ID parts unchanged from input to output assemblies.

After training, a number of separate *instances* of the prototype words are created by concatenating a unique ID with the content part of the developed prototype. These new words (e.g. *John*, *Mary*, *Bill*) are then added to the lexicon, and they can be used for input/output.

This *ID+content* technique is important for four reasons: (1) it makes it possible to deal with a large and open-ended set of semantically equivalent names without confusing them. In other words, we can create several tokens from the same basic type, and the system will know how to process them. This seems to be a prerequisite for modeling symbolic thought and logical reasoning (section 12.6.2). (2) Even if the system has been trained with only a small number of distinct meanings, it can process (e.g. *parse*) a much larger vocabulary *approximately*. The representations for the new words are cloned from the existing meanings (section 4.5.4). (3) New meanings can be incrementally learned, by using the cloned representation as the initial guess and letting it acquire finer semantic content in subsequent training (section 13.3.1). (4) It allows us to model creation of temporary and fictitious characters in story understanding. For instance, *John* in the beginning of the story is simply an instance of *human* (yet separate from other humans), until we learn more about him.

The ID part does not need to have any intrinsic meaning in the system. It is a surface-level tag, whose function is to distinguish the word from all other similar words. However, there are two sources from which the ID pattern could arise. (1) It could be based on the acoustical or orthographical qualities of the word itself. For example, even though *pot* and *pan* may have nearly identical meanings in a kitchen context, at the surface they are acoustically and orthographically distinct, and can be kept separate. (2) Or, it could be based on the sensory properties of the word referent. For example, the *man-who-hit* would have different image attached to it than the *man-who-helped*. For this reason, the *ID+content* technique can be thought of as an approximation of sensory grounding (section 12.6.2).

4.5.3 Performance with cloned synonyms

The *ID+content* technique was tested by developing a prototype word representation for each of the word equivalence classes: *human* for {*man*, *woman*, *boy*, *girl*}, *utensil* for {*fork*, *spoon*}, *predator* for {*wolf*, *lion*}, *glass* for {*plate*, *window*}, *gear* for {*ball*, *hatchet*, *hammer*}, *block* for {*paperwt*, *rock*} and *food* for {*cheese*, *pasta*, *carrot*}. These words (in braces) were replaced by their prototype in the training data and the identical sentences were removed. The new training set consists of 207 different sentences (see training code in appendix E.1.1 and data in appendix E.1.4).

After training, each prototype was cloned to cover the original vocabulary, choosing binary values for the two ID units (appendix E.1.5). The resulting set of representations is

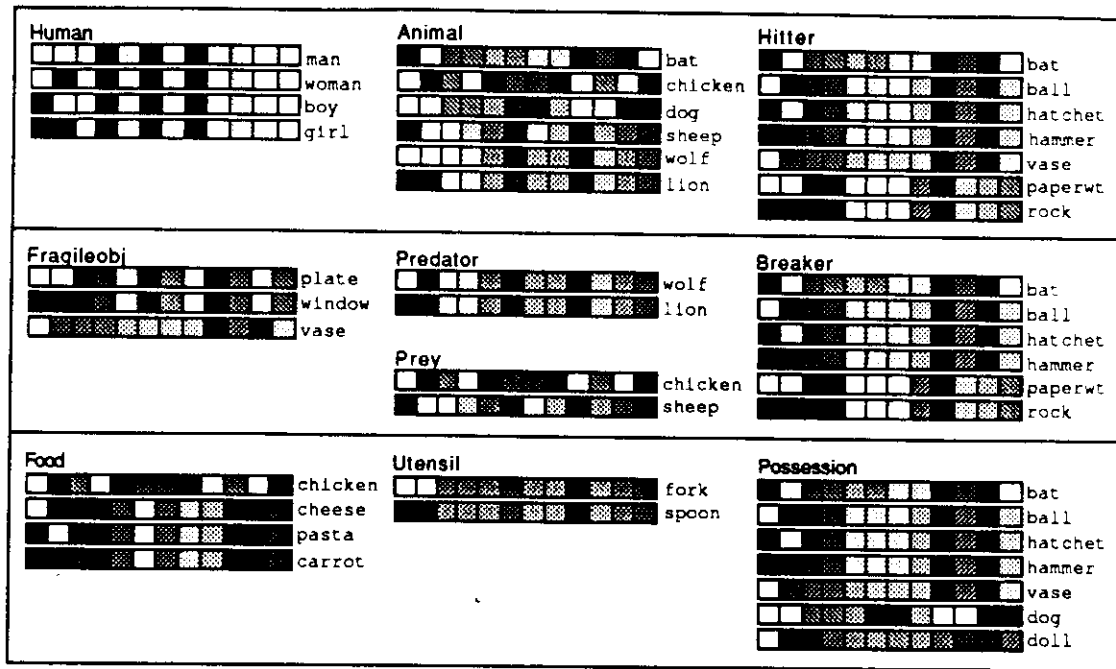


Figure 4.14: Final representations (cloned synonyms). The representations for the synonymous words {man, woman, boy, girl}, {fork, spoon}, {wolf, lion}, {plate, window}, {ball, hatchet, hammer}, {paperwt, rock} and {cheese, pasta, carrot} were formed by cloning a single prototype word. The first two units of the representation were used for the ID.

shown in figure 4.14. The representations reflect the categorization as before, and have the same processing characteristics. However, the members within each category are now more distinct.

Using the cloned representations the system is able to keep the equivalent words separate, as can be seen from table 4.6. The error in the output layer has not increased significantly, but 97% of all output words are now correct, with 98% of the words with synonyms.

With this particular task and data, the representations of words in neighboring equivalence classes, such as **gear** and **block**, become very similar. If the ID-patterns within the categories are very dissimilar, items are sometimes confused with items in close-by categories having the same ID-pattern. In other words, there is a danger that the ID is used as a basis for generalization. This can be avoided by keeping the ID part small and the ID-patterns different for different prototypes, e.g. random.

4.5.4 Extending the vocabulary

The ID+content technique can be applied to any word in the training data, and in principle, the number of clones per word is unlimited. This technique allows us to tremendously increase the size of the vocabulary while having only a small number of *semantically* different words at our disposal. Even though in principle each word has a unique meaning, this allows us to *approximate the meanings of a large number of words by dividing them into equivalence*

Data	All words	Instances	$E_i < 0.15$	E_{avg}
Familiar	96	96	95	.043
Unfamiliar	97	98	94	.045
Complete	97	98	96	.038

Table 4.6: **Performance of FGREP with cloned synonyms.** The network was trained with 0.1 learning rate for 2000 epochs and with 0.05 until epoch 5000, which took approximately the same time as training without cloning. The same test sets were used as in table 4.5, but now the synonymous words were represented by word instances created from the same prototype word. The average errors are slightly higher partly because the ID components of the instances consisted of extreme values 0 and 1, which are harder to achieve than midrange values.

classes.

This capability is important from the practical point of view, when we are trying to build an AI system to process natural language in the real world. Training the system with a large vocabulary is expensive. All the fine differences in usage of e.g. **give** and **donate** need to be included in the training examples. The ID+content technique allows us to train the system with the meanings which are most crucial for the task, and build the larger vocabulary (that the system needs to deal with in the real world) by expanding on those few basic meanings.

For example, the system can process sentences like **John** **downed** the lasagna with a **plastic-fork** and **Mary** **ate** the **chicken-soup** with a **silver-spoon**. The fine differences in meaning between these two sentences are available only to the external observer. The system would not understand the different connotations of these words, because it was not trained with them. Internally, the system processes only one event: **Human ate the food with a utensil**. While this is not exactly right, it captures the essential content of the sentences. This approach allows the system to perform its task (e.g. answer questions) robustly even if it does not capture all the subtleties of the words.

A similar approach is in fact taken by NLP systems based on conceptual dependency theory [Schank and Abelson, 1977]. For example, both **run** and **walk** map to a hand-coded structure involving the primitive act PTRANS, which stands for physical transfer. This approach allows a symbolic NLP system to handle many inputs, at the cost of losing the subtle connotations of similar words.

However, the FGREP with ID+content -approach also allows the connotations for similar words to be learned. A small initial vocabulary allows us to *bootstrap* the system with artificial training data. We can then extend the vocabulary with new instances of the basic meanings, and let them acquire more accurate semantics through FGREP adaptation in the actual task (section 13.3.1).

Figure 4.15 shows the performance data with an increasing number of clones for each word. The system was trained with the same data as before, but this time all words (except the blank) were cloned. In this experiment the ID values were spaced out evenly in the unit square (see appendix E.1.5). Uniformly distributed random values produced similar, only slightly weaker results. The space around each ID decreases inversely proportional to the

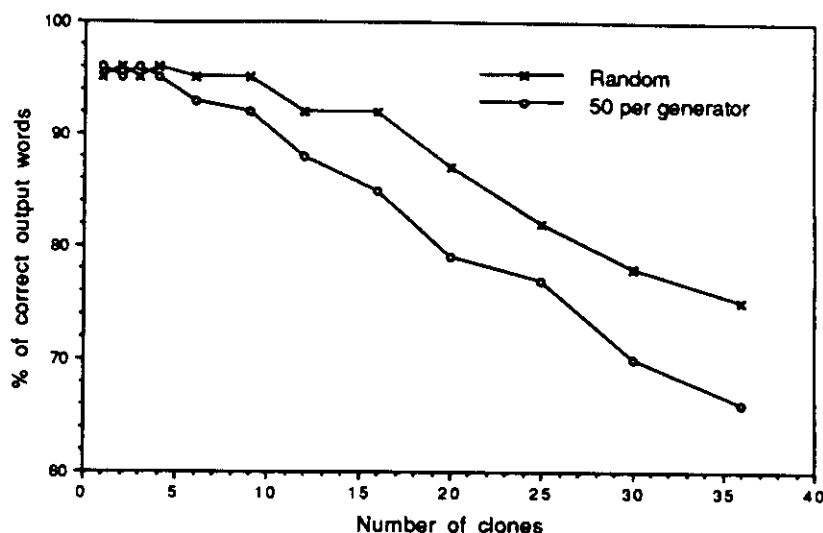


Figure 4.15: **Word discrimination of basic FGREP with extended vocabulary (ID+content technique).** The horizontal axis shows the number of clones created for each word, and the vertical axis indicates the percentage of correct output words. The Random -data set consisted of 1000 sentences selected randomly among the set of all sentences, while the 50 per generator -data set consisted of an equal number of randomly chosen sentences from each generator. 0.1 learning rate was used for the first 3,000 training epochs and 0.05 for another 3,000. The error average is not affected by the number of clones. The average error for the Random -data set was 0.035, with 97% of the output within 0.15, and for the 50 per generator -data set 0.040 and 95%.

number of clones. In other words, as the number of clones increases, the IDs become more and more similar, making the discrimination harder.

As can be seen from the figure, discrimination degrades approximately linearly as a function of clones. With sixteen different clones represented in 2-unit IDs the system still produces correct words 93% of the time. This is remarkable since *the number of different sentences grows polynomially, proportional to the fourth power of the number of clones*. For a single clone, there are 210 different sentences, four clones produce 27,024, sixteen 5,495,040 and thirty-six clones give us 134,401,680 different sentences.

Cloning word instances is very much like generating new symbols with the LISP function gensym. In addition, *the instances automatically have intrinsic meaning coded in them*. The processing knowledge is separate from the symbols that can be processed. With linear cost, the system can process a combinatorial number of inputs [Brousse and Smolensky, 1989] in a nontrivial task.

4.6 Processing sequential input/output: The recurrent FGREP module

The sentence processing architecture presented in the preceding sections relies on highly preprocessed input. An external supervisor determines the syntactic constituents of each

sentence, which is a nontrivial task in itself, and the sentence is represented in terms of these constituents in a fixed assembly-based representation.

In this section we show how these requirements can be relaxed. A recurrent extension of the FGREP architecture is presented, allowing us to efficiently deal with sequential input data. The architecture is used to form case-role representations directly from word-by-word sequential natural language input without the need for preprocessing.

4.6.1 Representing constituent structures

When the data is represented with *role-specific assemblies*, as in [Hinton, 1981; McClelland and Kawamoto, 1986] and in the FGREP architecture presented so far, the constituents of a complex data item can be correctly interpreted only in their appropriate assemblies. An assembly must be reserved for each possible constituent, whether it is actually present in the input or not. For example, there is an “object” section and a “with” section in each sentence representation, even though all sentences do not have these elements. Representation of structure becomes a serious problem when we want to scale up and represent, e.g., stories. The number of different constituents is enormous although only a few of them are present at any one time. Separate assemblies have to be reserved for each one of them. This leads to a combinatorial explosion in the size of the representations.

On the other hand, role-specific assemblies preserve the high-dimensional relations of the constituents. The relation of a constituent to all other constituents is well-defined as soon as it is placed in a specific assembly. All constituents of the representation are available simultaneously and in parallel, which makes efficient higher-order inferencing possible. This powerful representation style is suitable and desirable for internal representation, which is usually not severely limited by bandwidth.

Assembly-based representation can sometimes be made more efficient by letting part of the representation determine how the rest of the assemblies should be interpreted. The representation consists of two parts: the *base-part* determines the semantics of the *body* of the representation. This use of *data-specific assemblies* makes the most sense when the data consists of disjoint classes with distinct constituent roles, such as stories based on different scripts (see section 5.1). The technique reduces the number of assemblies needed, while still providing the same representational power.

Input/output channels, however, typically have very limited bandwidth. In certain cases, some of the assembly-based representational power can be given up to achieve more efficient communication. If the relations are sufficiently simple, information about partial ordering of the elements might be enough.

If there is a plausible way to linearize the data structure, representing it as a sequence is an efficient solution. There is no need to represent missing constituents, and structural information is conveyed by the order of constituents. In other words, I/O of complex data takes place exactly as in natural language, where complex thoughts are transformed into a sequence of words for transmission through the narrow communication channel. Consequently, natural language input/output is most naturally and efficiently represented in this manner.

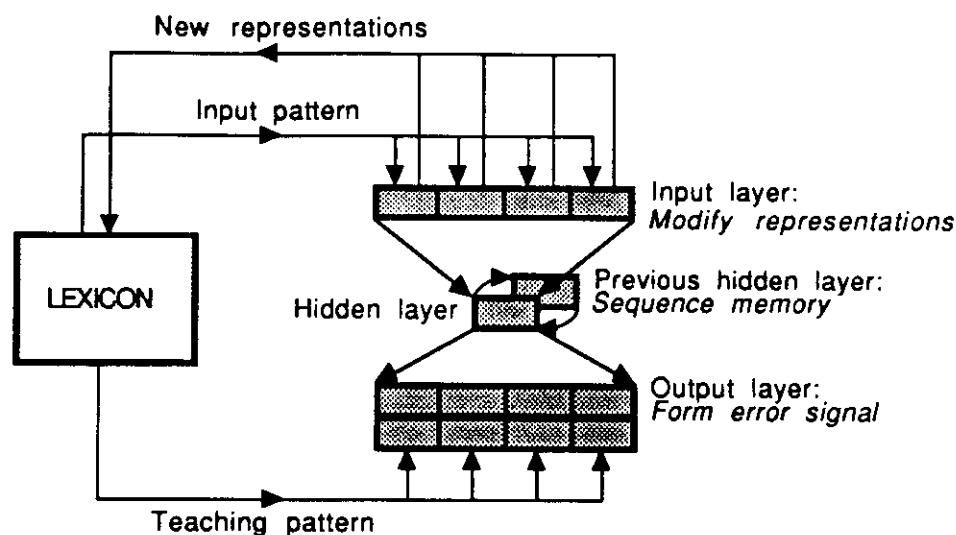


Figure 4.16: **Recurrent FGREP-module.** The hidden layer pattern is saved after each step in the sequence, and used as input to the hidden layer during the next step, together with the actual input.

4.6.2 Recurrent FGREP module

In recurrent FGREP, i.e. the extension of FGREP to sequential input and output [Miikkulainen and Dyer, 1989b], the basic network architecture is similar to the simple recurrent network proposed in [Elman, 1990; Elman, 1989] (see also [Jordan, 1986; Servan-Schreiber et al., 1989; St. John and McClelland, in press]). A copy of the hidden layer at time step t is saved and used along with the actual input at step $t + 1$ as input to the hidden layer (figure 4.16). The previous hidden layer serves as a sequence memory, essentially remembering where in the sequence the system currently is and what has occurred before. During learning, the weights from the previous hidden layer to the hidden layer proper are modified as usual according to the backpropagation mechanism.

The recurrent FGREP module can be used both for reading a sequence of input items into a stationary output representation, and for generating an output sequence from a stationary input. In a *sequential input network*, the actual input changes at each time step, while the teaching pattern stays the same. The network is forming a stationary representation of the sequence. In a *sequential output network*, the actual input is stationary, but the teaching pattern changes at each step. The network is producing a sequential interpretation of its input. The error is backpropagated and weights are changed at each step. Both kinds of recurrent FGREP networks are also simultaneously developing representations in their input layers (see the code in appendix C.1.1).

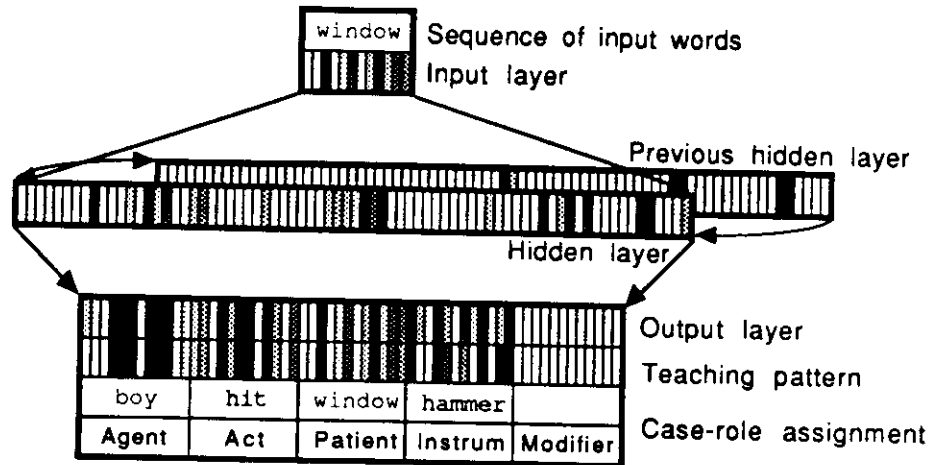


Figure 4.17: Snapshot of recurrent FGREP simulation. The system is in the middle of reading *The boy hit the window with the hammer.*

4.6.3 Case-role assignment from sequential input

Recurrent FGREP was tested with the same case-role assignment task and the same sentence data as before, with cloning of synonymous words. The network architecture is illustrated in figure 4.17. More resources are required to assign case-roles from sequential input than in the preprocessed case. The hidden layer now serves also as a sequence memory, and more capacity is needed. The size of the hidden layer was increased to 75 units. Training times are longer, because the sequences must be learned also, and backpropagation is done after each item in the sequence.

The input consists of the actual words of table 4.1, including now the words *the* and *with* (see appendix E.1.6). The main purpose was to show that the network can process raw natural language, but the network also learns to use the redundant words to disambiguate expectations (section 4.6.4). Word by word, the representations are fetched from the lexicon and loaded into the single input assembly. The activity is propagated to the output layer. The activity pattern at the hidden layer is copied to the previous-hidden-layer assembly, and the next word is loaded into the input layer. The case-role representation of the sentence thus gradually forms at the output. Each sentence is ended with a period, *which forms its own representation in the lexicon*. The case-role representation is complete after the period is input.

The performance figures are presented in table 4.7. Processing word sequences does not significantly degrade the performance: the system now outputs 95% of the word instances correctly, with 93% of the output values within 0.15. The representations look very much like in the stationary case, with the same processing properties.

Data	All words	Instances	$E_i < 0.15$	E_{avg}
Familiar	96	96	94	.042
Unfamiliar	97	97	94	.043
Complete	97	95	93	.045

Table 4.7: **Performance with cloned synonyms and sequential input.** 0.1 learning rate was used during the first 1500 epochs, 0.05 until 2000 and 0.025 for another 100 epochs.

4.6.4 Sequential expectations

When the input is sequential, expectations embedded in the word representations become clearly demonstrated. The expectations arise in the assemblies for unspecified case-roles (figure 4.17). During training, the network is required to produce a complete output pattern after each step in the sequence, even before it is possible to unambiguously recognize the sequence. As a result, the output patterns at each step are averages of all possible event sequences at that point, weighted by how often they have occurred. After the next word is input, some of the ambiguities are resolved and correct patterns are formed in the corresponding assemblies. Often the sentence representation is almost complete before the sentence has been fully input.

In figure 4.17, the network has read **The boy hit the window**, and has unambiguously assigned these words to the agent, act and patient roles. The instrument and modifier slots indicate expectations. At this point it is already clear that the modifier slot is going to be blank, because only human patients can have modifiers in the data (table 4.1). Most probably an instrument is going to be mentioned later, and an expectation for a hitter (an average of all possible hitters) is displayed in the instrument slot. If **with** is read next, a hitter is certain to follow, making the pattern even stronger. Reading a period next instead would clear the expectation in the instrument slot, indicating that the sentence is complete without this constituent.

The expectations emerge automatically and cumulatively from the input word representations. This feature can be used e.g. to fill in missing input information (section 5.3.4), or to guess the meaning of an unfamiliar word (section 13.3.1).

4.7 Limitations

There are certain restrictions in the FGREP approach. An FGREP network codes the meaning of an input item into its representation, and the meaning can be demonstrated by creating expectations, as was shown in sections 4.4.5 and 4.6.4. However, the representations can be fully interpreted only by the network(s) that developed them. For the rest of the world, the representations for **sheep** and **wolf** look similar, but there is no way to tell that they are both animals. Just by looking at a representation it is impossible to extract the features coded in it.

The representations still stand for the similarities between concepts, and it is possible

to train other systems to use the same set of representations efficiently. They are optimal only in the task they were developed in, but the representations still form a fairly good basis for other tasks. The semantics of single representations are not portable, and must be re-developed in the new task.

The mechanism is also somewhat limited by the fixed assembly-based internal representation. It is possible to use sequential input or output, but the internal representation must be laid out on a fixed number of assemblies. The assemblies can be data-specific (see section 4.6.1), which significantly increases their representational power. However, multiple fillers are still a problem. Sentences like *John, Mary, Bill and Susan went to the beach* cannot be represented on a fixed number of slots without posing a hard limit on the number of possible agents.

Above, the FGREP mechanism was demonstrated in parsing simple sentences. These sentences could be processed by a single recurrent FGREP module and they could be easily represented in the case-role assignment form. More complex parsing with a single module is also possible, but it is better to build a parser from several modules. For example, two hierarchically organized modules can read sentences with multiple hierarchical relative clauses into a canonical internal representation [Miikkulainen, 1990b] (this architecture is described in section 13.1.3). Building from FGREP modules is a powerful technique, and it is discussed in detail in the next chapter.

Chapter 5

Building from FGREP modules

The FGREP modules were designed to be used as building blocks in more complex cognitive systems. DISCERN contains six of such modules: the parsers read sequential input, the generators produce sequential output and the question answering modules are nonrecurrent FGREP modules. In this chapter we show how these modules can be put together in a hierarchical organization to implement a higher-level task, and analyze the performance of DISCERN without memory.

5.1 Performance phase

During performance, the whole system is a network of six subnetworks, each feeding its output into the input layer of another network. The input and output of each network consists of distributed representations of words, which match the ones stored in the global lexicon. Let us present the system with the example story:

John went to MaMaison. John asked the waiter for lobster. John left a big tip.

The task of the sentence parser network is to form a stationary case-role representation of each sentence, as has been described in the previous chapter. Words are input to this network one at a time, and the representation is formed at the output layer (figure 5.1). After a period is input, ending the first sentence, the final activity pattern at the output layer is copied to the input of the story parser network. The story parser has the same structure as the sentence parser, except that it receives a sequence of the sentence case-role representations as its input, and forms a stationary slot-filler representation of the whole story at its output layer.

The final result of reading the story is the slot-filler assignment at the output of the story parser network. This is the representation of the story in terms of its role bindings, with two additional assemblies specifying the script and the track. The role assemblies stand for different roles depending on the script. In our example, the representation consists of script=restaurant, track=fancy, customer=John, restaurant=MaMaison, food=lobster, taste=good and tip=big. This technique makes the best use of the fixed-size assembly-based representation. The assemblies are data-specific, rather than role-specific, since their interpretation varies with the data. The script assembly constitutes the base of the representation. For example, if script=\$travel, then the fourth assembly (R/food in figures 5.1 and 5.2) will serve the role of T/origin, the origin of the trip.

The internal representation completely specifies the events of the script. The generating subsystem (figure 5.2) is trained to paraphrase the story from the internal representation.

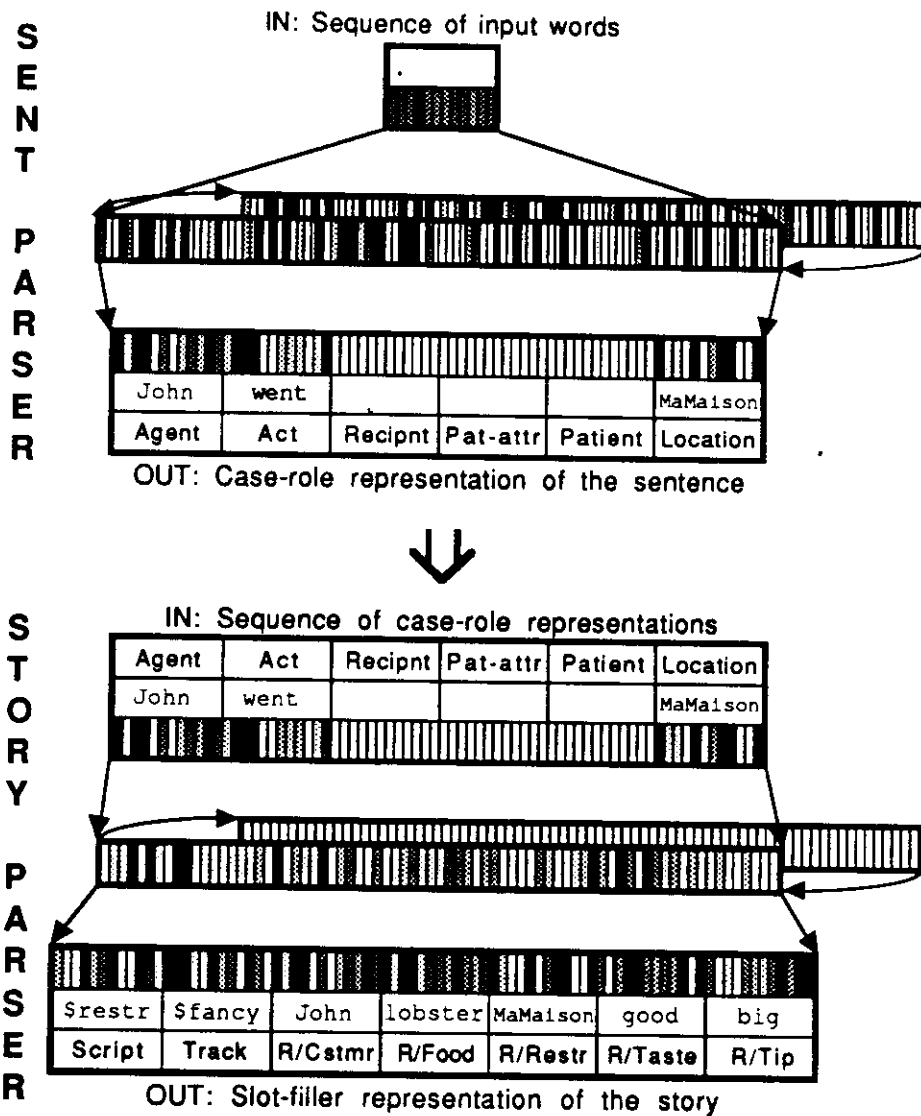


Figure 5.1: **Networks parsing the story.** The figure presents a snapshot of the simulation after the first sentence of the example story has been read. A period, ending the first sentence, is in the input assembly of the sentence parser. The script and the track have already been identified automatically and a number of expectations about the role bindings are active at the output of the story parser. The role names (R/...) are specific for the restaurant script.

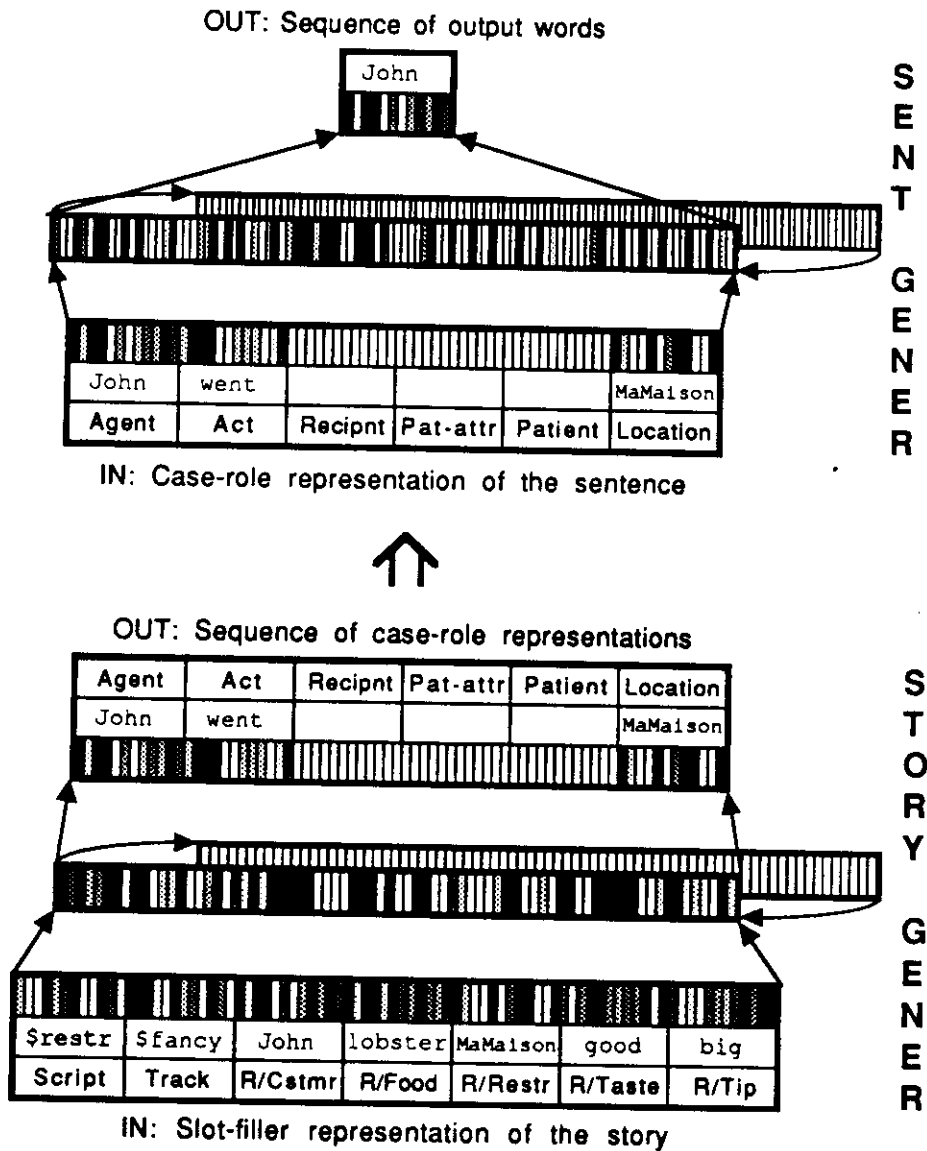


Figure 5.2: **Networks generating the paraphrase.** Snapshot of the simulation, where the story generator has produced the case-role representation of the first sentence, and the sentence generator has output the first word of that sentence. The previous hidden layers are blank during the first output.

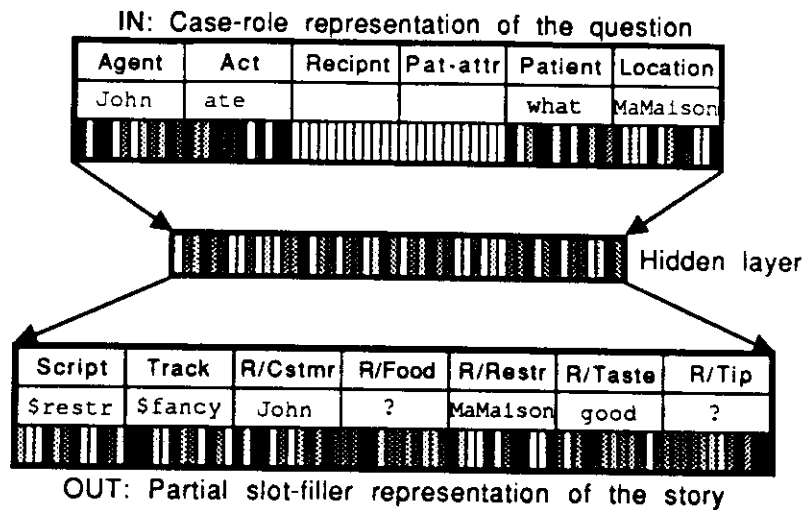


Figure 5.3: The cue former module. The module takes a case-role representation of the question as its input and produces an incomplete slot filler representation of the story in question as its output, where unspecified slots indicate weighted averages of all possibilities.

The idea is simply to reverse the process of reading in.

The story generator network (figure 5.2) receives the complete slot-filler representation of the story as its input, and generates the case-role assignment of the first sentence of the story as its output. This output is fed into the sentence generator network, which produces the distributed representation of the first word of the first sentence as its output. Again, the hidden layer of the sentence generator network is copied into the previous-hidden-layer assembly and the next word is output.

After the last network produces a period, indicating that it has completed the sentence, the hidden layer of the story generator network is copied into its previous-hidden-layer assembly, and the story generator network produces the case-role representation of the second sentence. The process is repeated until the whole story has been output.

The sentence parser and sentence generator networks are also trained to process question sentences and answer sentences. The cue former module has the high-level task of determining what story in the memory the question is referring to. This is accomplished by a non-recurrent FGREP-architecture (figure 5.3). Suppose the system is asked, What did John eat at MaMaison? The sentence parser network reads the question sequentially word by word, and forms a case-role representation of it. The cue former takes this representation and determines that in the story this question is referring to, the script-slot must be \$restaurant, customer=John, restaurant=MaMaison, track=\$fancy (because MaMaison is a fancy-restaurant) and taste=good (because food is always good in fancy-restaurants in our data). A partial story representation is formed from these elements, with unknown slots indicating averages of all possible alternatives, weighted by their probabilities. In complete DISCERN, this pattern is used to cue the memory for the complete representation of the story.

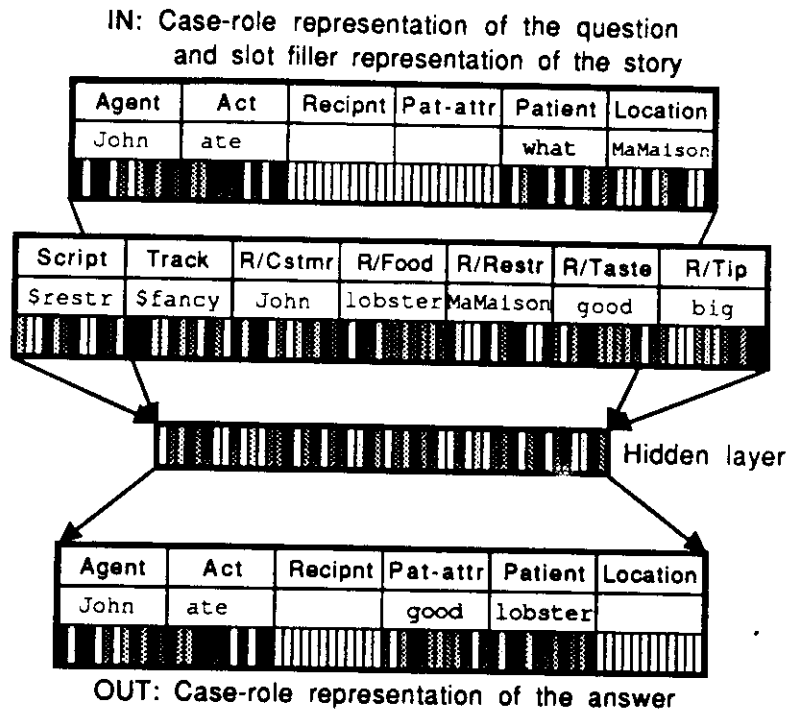


Figure 5.4: **The answer producer module.** The module takes a case-role representation of the question and the slot filler representation of the appropriate story as its input, and produces the case-role representation of the answer as its output.

The answer producer is implemented as another non-recurrent FGREP module (figure 5.4). The network receives as its input the representation of the question, together with the slot-filler representation of the story. In complete DISCERN this representation is obtained from the episodic memory; in the analysis in this chapter, the representation is simply obtained by concatenating the correct word representations from the lexicon. The network generates a case-role representation of the answer sentence, which is then output word by word with the sentence generator network.

5.2 Training phase

A good advantage of the modular architecture can be made in training these networks. The tasks of the six networks are separable, and they can be trained separately (e.g. on different machines) as long as compatible I/O material (i.e. the same stories and the same lexicon) is used. The networks must be trained simultaneously, so that they are always using and developing the same representations.

The lexicon ties the separated tasks together (figure 5.5). Each network modifies the representations to improve its performance in its own task. The pressure from other networks modifies the representations also, and they evolve slightly differently than would be the most efficient for each network independently. The networks compensate by adapting their

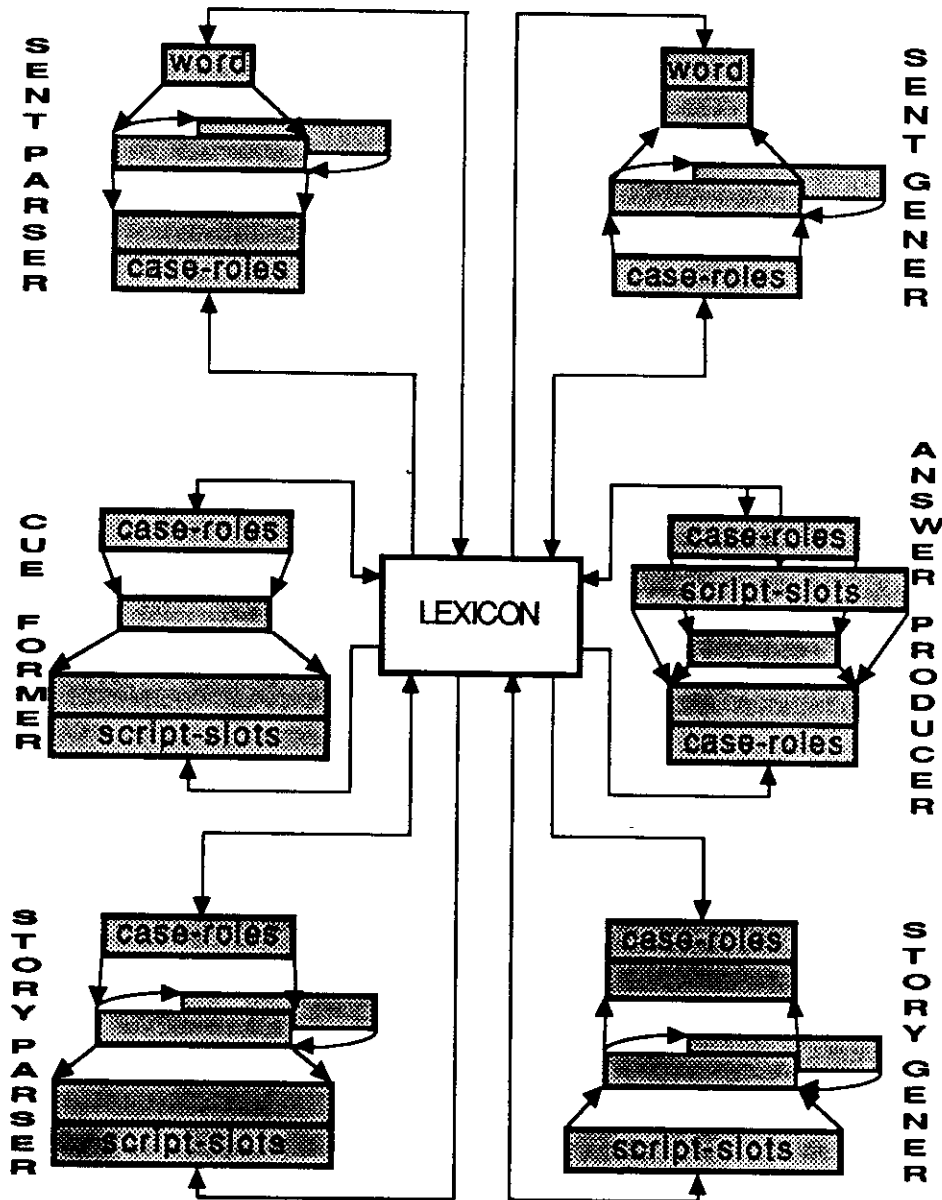


Figure 5.5: **Training the FGREP modules.** Each network is trained separately and simultaneously, developing the same lexicon. Words are taken from the lexicon and placed in assemblies in the input and output layers of each network. Modified word representations (via FGREP method) are placed back in the lexicon and used again by other networks, until all words representations stabilize for all training data for all networks.

weights, so that in the end the representations and the weights of all networks are in harmony. *The requirements of the different tasks are combined, and the final representations reflect the total use of the words.*

If the training is successful, the output patterns produced by one network are exactly what the next network learned to process as its input. But even if the learning is less than complete, the networks perform well together. Erroneous output patterns are noisy input to the next network, and neural networks in general tolerate, even filter out noise very efficiently.

5.3 Processing modules in DISCERN

5.3.1 Training data

The DISCERN system was trained to paraphrase and answer questions about stories based on restaurant, shopping and travel scripts. There were three tracks to each script, with four to five open roles and four different questions. The story and question/answer skeletons for each track are listed in appendix A.

The actual stories and questions are formed from these skeletons by specifying the ID part for each prototype word that appears in the skeleton. In the training, the IDs for these words are assigned randomly for each story, but consistently throughout the whole story. The test stories and questions/answers were generated by creating two instances from each prototype. The system was tested with two test sets: (1) complete stories, which included all sentences in the skeleton, and (2) incomplete stories, containing only three sentences. The same questions were used in both cases.

For example, a test story where the prototype word PERSON has been instantiated with John, FANCY-FOOD with lobster and FANCY-RESTAURANT with MaMaison reads:

John went to MaMaison.*
The waiter seated John.
John asked the waiter for lobster.*
John ate a good lobster.
John paid the waiter.
John left a big/small tip.*
John left MaMaison.

The incomplete version of this test story includes only the sentences marked with "*". Questions/answers for this story are:

Q: Who ate lobster at MaMaison?
A: John did.
Q: What did John eat at MaMaison?
A: John ate a good lobster.
Q: Where did John eat lobster?
A: John ate lobster at MaMaison.

Q: How did John like lobster at MaMaison?
A: John thought lobster was good at MaMaison.

The need for the script and track assemblies in the internal representation is obvious from the data. The script is used to specify how the patterns in the role assemblies should be interpreted, i.e. what the roles are. The representation is data-specific, depending on the pattern in the script-slot for its interpretation. The track is necessary because the order of events is slightly different in different tracks. Without script and track slots it would be necessary to represent the differences as additional role bindings, which would be inefficient and unnatural. The script and track patterns are treated just like words in the system. Their representations are stored in the lexicon and modified by FGREP during training.

The restaurant tracks in the training data contain some interesting regularities: the food is always good in a fancy-restaurant, and always bad in a fast-food restaurant. In coffee-shop-restaurant, the size of the tip correlates with the food: good food \Leftrightarrow big tip, bad food \Leftrightarrow small tip. These regularities were set up intentionally; the system should be able to use them to fill in unspecified roles.

5.3.2 Simulation set-up

The six networks were trained separately and simultaneously with compatible I/O data. Each story was processed as if the networks were connected in a chain, but the output patterns, which are more or less incorrect during training, were not directly fed into the next network. Instead, they were replaced by the correct patterns, obtained by concatenating the current word representations taken from the lexicon. This way each network was trained over the same number of epochs with compatible data. For an alternative training method, where the networks are trained on separate machines, see [Miikkulainen and Dyer, 1989b] and the code in appendix C.1.1.

The learning rate was gradually reduced from 0.1 to 0.005 over a total of 25,000 epochs (0.1, 0.05, 0.025, 0.01 and 0.005 for 5,000 epochs each). Word representations consisted of 12 units. The sentence parser and generator had 100 units in their hidden layer, the story parser and generator had 75, and the cue former and answer producer had 50. Two units were used for the IDs. Key parts of the training code and data, code for generating the data and simulation specification file are listed in appendix C.1.

In the performance phase two instances were created from each prototype word, using orthogonal values for the IDs (1,0 and 0,1). The set of test stories was put together from all combinations of the instances, 96 stories altogether (including the two different versions of tip and distance for the fancy-food, coffee-shop-food and train-travel tracks). The performance of the system was tested in paraphrasing and answering questions about complete and incomplete input stories. The performance code, samples of the test data and code for generating the test data are listed in appendices D.1.3, D.3 and D.2.

Network	All words	Instances	$E_i < 0.15$	E_{avg}
Sentence parser	99	99	98	.018
Story parser	100	100	98	.017
Story gener	100	100	100	.015
Sentence gener	100	100	99	.028

Table 5.1: **Performance of the parser and generator networks in isolation.** Two instances for each prototype word were cloned, generating a total of 96 different test stories.

5.3.3 Representations

There are very few similarities in the resulting representations. The vocabulary in the example stories is fairly large compared to the number of different stories (99 word prototypes vs. 9 story skeletons), and most words have very distinct usage. As a result, the word representations also become distinct. Only the words for different types of restaurants, foods, shops, shopping items and travel destinations are faintly similar. The system has no trouble keeping the words separate with this data.

The system learns to output the period quite early in the training. Very seldom is a sentence produced without a period at the end. On the other hand, in the early stages of training, sentences are often ended prematurely with a period, and after that, only periods are output. This happens because the representation for the period is modified at the end of every sentence, where reading it in should have very little effect on the output. Its representation becomes the most neutral representation, i.e. it contains many components close to 0.5 (see e.g. figure 5.1). This representation makes the period a default output when the network does not know what word to generate next.

The period is treated in the system just like a word, with a semantic representation of its own, formed automatically from the input examples. It seems that this approach could be used to deal with the semantics of punctuation in general. The network develops a representation for each punctuation symbol according to how it is used. The representation encodes information about possible contexts, and affects how the rest of the input is interpreted.

5.3.4 Paraphrasing

The training corpus consisted of complete stories, and the system was trained to reproduce a story exactly as it was input. Table 5.1 presents performance figures for each network in their respective tasks in isolation. In this test there was no communication between modules, each network was tested separately in the task it was trained in. The input consisted of compatible but error-free story data.

Table 5.2 shows the performance of each network in the same task, but now the networks are connected in a chain, the output of one feeding to the input of the next network. Interestingly, the errors do not cumulate very much in the chain. Because the representations are distributed and redundant, the noise in the input is efficiently filtered out, and each network

Network	All words	Instances	$E_i < 0.15$	E_{avg}
Sentence parser	99	99	98	.018
Story parser	100	100	95	.022
Story gener	100	100	97	.017
Sentence gener	99	99	96	.033

Table 5.2: **Performance of the parser and generator networks connected in a chain.** The same story data was used as in table 5.1, but the input of the story parser and the story and sentence generators was obtained from the output of the previous network.

Network	All words	Instances	$E_i < 0.15$	E_{avg}
Sentence parser	99	99	98	.020
Story parser	95	94	92	.041
Story gener	98	95	96	.024
Sentence gener	98	93	95	.041

Table 5.3: **Performance of parser and generator networks with incomplete input stories and connected in a chain.** The same set of stories were used as in table 5.1, but only three sentences (marked with “*” in the story skeletons in appendix A) were included.

performs approximately at the same level. In the output story, 99% of the words are correct, including 99% of the cloned words. In other words, the system produces the right customer, food, store, destination etc. 99% of the time.

However, the clean-up does not apply to the errors in the ID units, because they are nonredundant. The rest of the representation gives no clue about what the value of the IDs should be, and an error made in an ID unit early in the chain cannot be corrected later. This means that the network is more likely to produce errors where the different instances of the same type are confused, rather than confusing words with different meaning. This side effect is interesting since it seems to match human behaviour. Role bindings based on syntactic constraints are harder than ones that are supported by a strong semantic component.

Given that the system was trained to reproduce its input story, an interesting question is how well the system can fill in the events when the input story consists of only a few sentences. The system performs very well in this respect (table 5.3). There is very little degradation in performance compared to the complete stories. The story parser introduces some error to the flow of information, but much of it is filtered out by the story generator. Incomplete stories which uniquely specify a story are paraphrased correctly to their full extent. The quality of the food (good or bad) is inferred if the size of the tip is mentioned in the story, and vice versa. If there is not enough information to infer the filler for some role, an activity pattern is produced which is intermediate (a weighted average) between the possible choices. This follows directly from the tendency to generate expectations.

For example, if the story consists of only the sentence John went to MaMaison, the food

quality can be inferred (good, because MaMaison is a fancy-restaurant), but an intermediate representation develops in the food-slot (figure 5.1). In paraphrasing the story it seems as if one of the possible fancy-foods is chosen at random. Especially interesting is, that once a role binding (e.g. food=**steak**) is selected in an earlier event, even if it is incorrect it is usually maintained throughout the paraphrase generation. The choice is consistent throughout the story, because all sentences are generated from the same pattern in the food-slot. Thus, DISCERN *performs plausible role bindings* – an essential task in high-level inferencing and postulated as very difficult for PDP systems to achieve [Dyer, 1990].

In general it seems that a network which builds a stationary representation of a sequence might be quite sensitive to omissions and changes in the sequence. Each input is interpreted against the current position in the sequence by combining it with the previous hidden layer. If items are omitted from the sequence it seems that the system could very easily lose track. But this is not the case in the script reader. The network was trained to always shoot for the complete output representation, and the hidden layer pattern stabilizes very early in the sequence. Reading subsequent input items has very little effect on the hidden layer pattern, and consequently, *omissions are not critical*.

Filling in the missing items is a form of generalization. A similar generalization in a non-sequential network (e.g. such as described in the earlier sections) would be required when a number of input assemblies are fixed at 0.5, the “don’t care” value. The strong filling-in capability of DISCERN is due to the fact that there is very little variation in the training sequences. But it is exactly this lack of variety in the sequence which makes up scripts – they are stereotypical sequences of events. Interestingly, it follows that *filling in the unmentioned events is an easy task for a sequential network system* such as DISCERN.

Once the script has been instantiated and the role bindings fixed there is no way of knowing which of the events were actually mentioned in the story. What details are produced in the paraphrase depends on the training of the output networks. This result is consistent with psychological data on how people remember stories of familiar event sequences [Bower et al., 1979]. The distinction of what was actually mentioned and what was inferred becomes blurred. Questions or references to events which were not mentioned are often answered as if they were part of the original story.

5.3.5 Question answering

Question answering (besides paraphrasing) is another standard task for an NLP system, which can be used to demonstrate the system’s level of understanding. Question answering in general is a complex task of its own [Lehnert, 1978], however, DISCERN does not get very deep in the issues in the task. The questions are simple inquiries about the role bindings in the story, such as **What did John eat at MaMaison?**, **Where did Mary take a plane to?**, which are used to demonstrate that DISCERN has built a meaningful internal representation of the story. Often answering questions requires inferences beyond the original incomplete story. In this respect question answering is another illustration of script-based inference, similar to paraphrasing. On the other hand, the task shows how the modular FGREP architecture can be extended by adding another task, implemented with additional modules.

Network	All words	Instances	$E_i < 0.15$	E_{avg}
Sentence parser	98	94	97	.025
Cue former	81	80	89	.051
Answer producer	100	100	100	.013
Sentence gener	100	100	98	.028

Table 5.4: **Performance of isolated sentence processing and question answering modules in the question answering task.** The same set of stories was used as before, now only the questions and answers were processed.

Network	All words	Instances	$E_i < 0.15$	E_{avg}
Sentence parser	98	94	97	.025
Cue former	80	76	86	.060
Answer producer	100	100	96	.019
Sentence gener	100	99	95	.036

Table 5.5: **Performance of connected modules in question answering.** The story representation input of the answer producer was obtained from the story parser, table 5.2

DISCERN was trained with **who**, **what** and **where** -questions, and also with questions like **Did John travel a big distance?** and **How did John like the lobster in MaMaison?** DISCERN can be trained with any question type, and as long as it is reasonable (i.e. it can be answered using the information the network has about the script and the role-bindings) the network will learn to answer it. An interesting issue is, can the network generalize question types between different scripts? The prospects for this are outlined in section 11.7.

Tables 5.4, 5.5 and 5.6 present the performance of DISCERN in answering questions about incomplete stories. The figures show the performance of the same sentence parser and sentence generator network used for tables 5.1, 5.2 and 5.3, but now in processing question and answer data. The output of the cue former was compared to the target story representation, i.e. to the complete representation of the story that the question referred to. Because some of the slots are unspecified in the cue, the performance numbers for this network are lower than for the other networks. In tables 5.4 and 5.5, where the networks are connected, the complete story representation for the answer producer was obtained from the output of the story parser, the accuracy of which is shown in tables 5.1 and 5.2.

The sentence parser and generator have learned to process questions and answers, and the answer producer has learned to produce the answer case roles. The sentence parser and generator show slightly less accurate performance than in paraphrasing, because the question and answer sentences are more complex and specify more token information than the story sentences. That is, the ratio of errors/information is about the same in the two tasks.

The performance of the cue former is not obvious from the tables. A more careful inspection of the errors reveals that it quite accurately generates the average of all possibilities.

Network	All words	Instances	$E_i < 0.15$	E_{avg}
Sentence parser	98	94	97	.025
Cue former	80	76	86	.060
Answer producer	98	94	96	.025
Sentence gener	97	92	94	.043

Table 5.6: Performance in answering questions about incomplete stories.

This occurs because the cue former was required to shoot for the complete story representation for each example during training. Because the same question can refer to several stories in the training, the network learns to develop an average, weighted by how often each story occurred in the training data. This results in an interesting effect, discussed in section 11.7: the most common stories are easier to recall than seldom occurring ones. The cue for the common stories is more accurate, the rare story needs to be specified to more detail.

Part III

MEMORY MECHANISMS

Backpropagation-based network architectures, such as the processing modules in DISCERN, are very powerful for simple mapping tasks. The learning mechanism is extremely strong, and allows extensions such as the reactive training environment of FGREP, the ID+content mechanism and sequential I/O. With careful design of input and output and suitable preprocessing it can be applied to various processing tasks.

However, backpropagation networks do not lend themselves well to modeling long-term memory. There are three main problems:

(1) When presented with ambiguous input, they generate output where the different alternatives are blended. Sometimes this is appropriate, since memory confusions like **blue square, red circle** → **blue circle** do occur. Blending is not always plausible, though, since much of the memory recall is categorical. When reading an ambiguous word such as **bat**, humans do not develop a representation which is a combination of a stick and a flying bat. The situation is more analogous to **necker-cube** perception, where we can switch our attention between the two interpretations, but a blend does not occur. Even if one of the alternatives is absolutely certain, given the correlations in the training data, the network output will have crosstalk from other cases [St. John, 1990]. In many cases perception and processing are categorical. An inaccurate input item is recognized as a member of a familiar category, and replaced by an accurate representation for that class. In backpropagation networks, noisy and inaccurate input patterns are processed robustly, but the noise is never completely filtered out.

(2) Backpropagation networks are not parallel at the knowledge level [Sumida and Dyer, 1989]. They can only process one item at a time; they cannot do many-to-many mappings. This is a problem in e.g. modeling lexical disambiguation. There is evidence that all alternative concepts are accessed simultaneously for a brief period [Swinney, 1979]. Representing the different alternatives simultaneously but distinctively would be desirable, yet the backpropagation network can only produce a single pattern, a blend of all possibilities.

(3) Knowledge can be stored in a backpropagation network only by repetitive presentation of the item. In addition, all other items must also be presented together with the new item to be stored, in order to avoid forgetting them. There is no known way to make the network learn with a single presentation of an item, as required for episodic memory.

Part III discusses the memory mechanisms in DISCERN. Self-organizing feature maps are first reviewed, and they are shown to overcome the above problems. Mechanisms for the episodic memory (hierarchical memory organization and one-shot storage), and for the lexicon (many-to-many mapping) are discussed in turn.

Chapter 6

Self-organizing feature maps

Self-organizing feature maps [Kohonen, 1982b; Kohonen, 1984] are (1) a theory of biologically plausible knowledge organization, and (2) a method for unsupervised learning. A topological layout of input data is formed in a self-organizing process.

6.1 Topological feature maps

A 2-D topological feature map implements a topology-preserving mapping from a high-dimensional input space onto a 2-D output space. The map consists of an array of processing units, each with N weight parameters (figure 6.1). The map takes an N -dimensional vector as its input, and produces a localized pattern of activity as its output. In other words, an input vector is mapped onto a location on the map.

Each processing unit receives the same input vector, and produces one output value. The response is proportional to the similarity of the input vector and the unit's weight vector. The unit with the largest output value constitutes the image of the input vector on the map. The weight vectors are ordered in such a way that the output activity smoothly decreases with the distance from the image unit, forming a localized response.

The weight vectors are tuned to specific items of the input space so that topological relations are retained. This means roughly that nearby vectors in the input space are mapped onto nearby units in the map. This is a very useful property, since the complex similarity relationships of the high-dimensional input space become visible on the map.

For example, the story representations in DISCERN are high-dimensional vectors, and their relations are hard to see just by looking at the representations. With topological feature maps it is possible to lay out the representation space on a 2-D area so that the similarities between them become evident (figure 6.2). There are three major areas on the map, corresponding to the three scripts. Each area is further subdivided according to the tracks. For example, all restaurant stories are mapped on the "restaurant" area, and fancy-restaurant stories are mapped on the "fancy" subarea. In addition, similar role bindings within the track tend to be clustered together.

Strictly speaking, if two stories are close in the input space, the map responds with similar activity patterns when these vectors are presented as inputs. If the dimensionality is reduced in the mapping, items close in the input space are not always mapped close. This is because the map tries to approximate the higher-dimensional space with a Peano-surface. For example, when trying to cover a 2-D space with a 1-D line, the line must curve back to the same area, and at these areas nearby points are mapped far apart on the line.

On the other hand, items close on the map are not always close in the input space either.

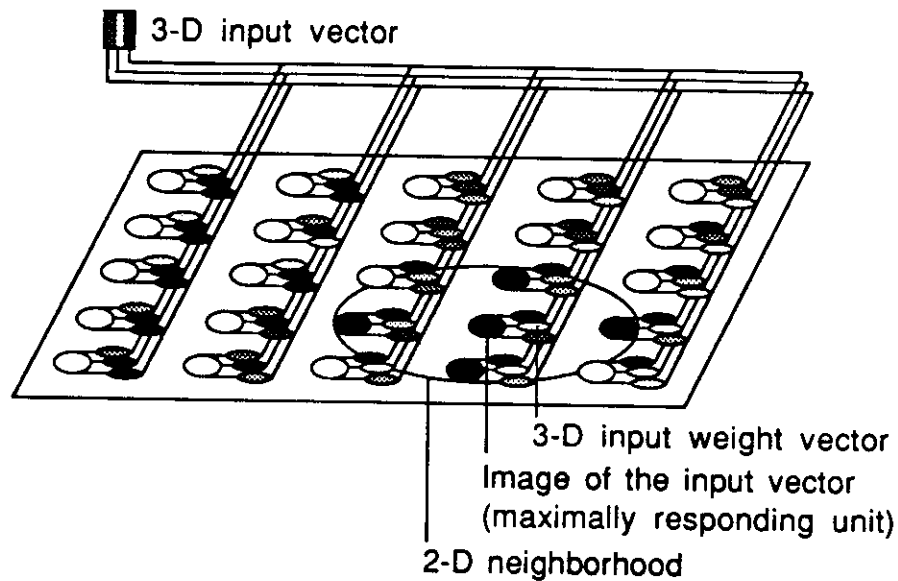


Figure 6.1: **A self-organizing feature map network.** In this example a mapping is formed from a 3-dimensional input space onto a 2-dimensional network. The values of the input components, weights and the unit output are indicated by gray-scale coding.

The mapping tries to cover the whole area of the map by data. For instance, the map of taxonomical data essentially displays the minimal spanning tree of the data items, curved to fill in the whole area of the map [Kohonen, 1982b]. Some items that are far in the taxonomy are bound to be mapped close by on the map, because no empty space is left between branches of the taxonomy. Or, if we form a mapping of script-based stories, variations of the same script are mapped near each other (figure 6.2), but so are e.g. coffee-shop, bus-travel and electronics shopping stories. There is no way of telling where the boundaries of the continuous areas are on the map.

To conclude, the map is topological only if one compares the response patterns of the whole map instead of just the image units. Even in terms of image units, the map does the best job possible in spreading a high-dimensional data on a 2-dimensional area. The map consists of subareas, which are continuous and where nearby units stand for nearby items in the input space. However, the boundaries of these continuous areas are not marked on the map.

6.2 Self-organization

6.2.1 General mechanism

The organization of the map, i.e. the assignment of the weight vectors, is formed in an unsupervised learning process [Kohonen, 1982a; Kohonen, 1982c; Kohonen, 1984; Ritter and Schulten, 1988]. Input items are randomly drawn from the input distribution and presented to the network one at a time (figure 6.1). The map responds to each vector by developing a

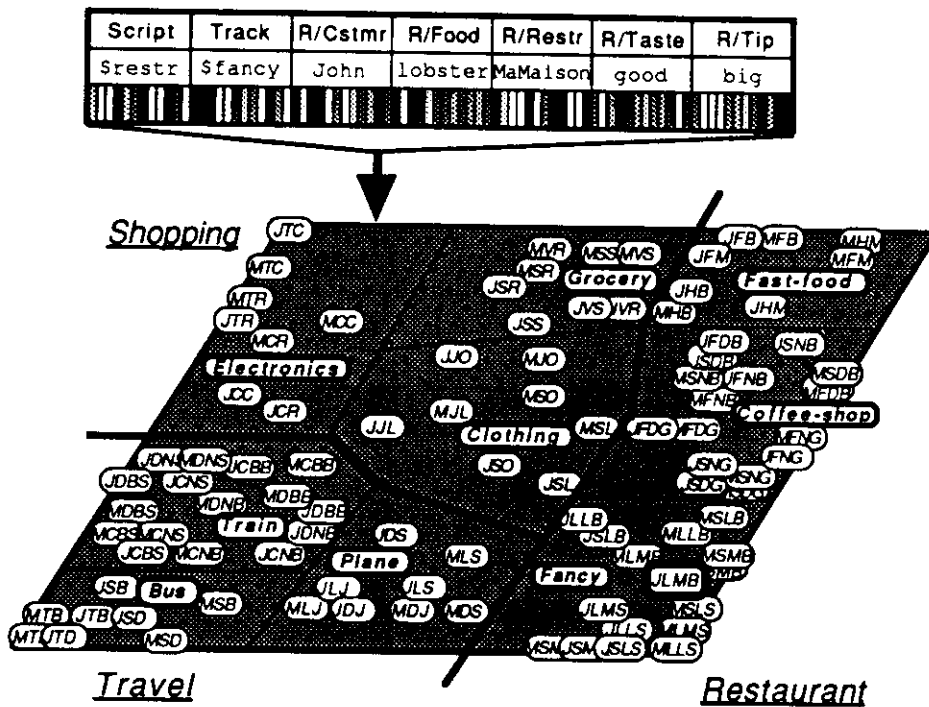


Figure 6.2: **Single-level mapping of script-based stories.** A story representation is presented to the map, and the maximally responding unit constitutes the image of the story on the map. For example, for the fancy-restaurant story in the figure, the unit JLMB on the map responds the strongest. The labels indicate image units for the 96 test stories used in testing DISCERN (see appendix A). The first letter of a label indicates the main actor in the story: J stands for John and M for Mary. The rest of the letters stand for the rest of the role bindings. For example in the fancy-restaurant area, the second letter stands for food (lobster or steak), third for restaurant (MaMaison or Leone's), and the fourth for tip (big or small). A complete explanation of the letters on these labels is presented in figures 7.6 to 7.8. For the simulation details, see section 7.3.2.

localized activity pattern. The weight vector of the maximally responding unit and each unit in its neighborhood are changed towards the input vector, so that these units will produce an even stronger response to the same input in the future.

In other words, the map adapts in two ways at each presentation: (1) the weight vectors become better approximations of the input vectors, and (2) neighboring weight vectors become more similar. Together these two adaptation processes eventually force the weight vectors to become an ordered map of the input space.

The process starts with very large neighborhoods, i.e. weight vectors are changed in large areas. This results in a gross ordering of the map. The size of the neighborhood is decreased with time, allowing the map to make finer and finer distinctions between items. Eventually, the distribution of the weight vectors becomes an approximation of the input vector distribution [Kohonen, 1982a; Kohonen, 1984]. This means that the most frequent areas of the input space are represented to greater detail, i.e. more units are allocated to represent these inputs. The two dimensions of the map do not necessarily stand for any recognizable features of the input space. The dimensions develop automatically to facilitate the best discrimination between the input items (figure 6.2).

Each adaptation step consists of three tasks: (1) measuring the similarity of the input vector and the unit's weight vector, (2) determining the adapting neighborhood, and (3) changing the weights within this neighborhood. These tasks can be implemented with different mechanisms. The theory originates from a biologically plausible implementation [Kohonen, 1982c], but much of the biological mechanisms can be abstracted and replaced with computationally more efficient algorithms without obscuring the process itself. Both approaches are briefly discussed below.

6.2.2 Biological model

In a biologically plausible model of self-organization [Kohonen, 1982c; Miikkulainen, 1987] the similarity is measured with a scalar product of the input vector and the weight vector, i.e. by computing a weighted sum of the input components. The neighborhood is selected by focusing the initial response of the map through lateral inhibition. The response $\eta_{ij}(t)$ of a unit (i, j) (in a 2-dimensional map) to an external input vector evolves over time according to

$$\eta_{ij}(t) = \sigma \left[\sum_h \mu_{ij,h} \xi_h + \sum_{k,l} \gamma_{kl,ij} \eta_{kl}(t - \Delta t) \right], \quad (6.1)$$

where $\mu_{ij,h}$ is the unit's h th weight component, ξ_h is the n th component of the input vector, $\gamma_{kl,ij}$ is the lateral connection weight on the connection from unit (k, l) to unit (i, j) , and $\eta_{kl}(t - \Delta t)$ is the activity of unit (k, l) during the previous time step. The function σ is the familiar sigmoid activation function of the type

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \quad (6.2)$$

The sigmoid introduces a nonlinearity (a soft threshold) into the response, and limits its output within $[0,1]$ range. The unit responds very weakly at first until it receives a certain amount of input activation, and the response saturates soon after that.

The weights on the lateral connections have the form of a “Mexican hat”, i.e. difference of Gaussians. The connections to the closest units are excitatory and to units further away are inhibitory. This is a biologically plausible form of lateral connectivity, well documented in e.g. low-level vision systems [Hartline, 1949; Ratliff et al., 1966]. The lateral weights are assumed to be preassigned, and their task is to support self-organization of the weights on the external input connections. The self-organizing process does not modify the lateral connections.

The primary effect of lateral inhibition is to sharpen the contrast between the high and low activity areas. The response of the network is focused around the maximally responding unit (or area) in successive iterations of equation 6.1. The more lateral inhibition there is compared to the lateral excitation, the more focused is the final response.

After the network has settled, the input weight vectors are rotated towards the input vector:

$$\Delta\mu_{ij,h}(t + \Delta t) = \frac{\mu_{ij,h}(t) + \alpha(t)\eta_{ij}(t)\xi_h}{\left\{\sum_h [\mu_{ij,h}(t) + \alpha(t)\eta_{ij}(t)\xi_h]^2\right\}^{1/2}}, \quad (6.3)$$

where $\alpha(t)$ is the gain of the weight change at time t .

The rotation of the weight vectors models redistribution of synaptic resources, where the synaptic efficacy (i.e. the weight) is proportional to the square root of the resource. However, the length of the weight vector must be preserved in the adaptation. If the network is to form a map of an N -dimensional input space, a redundant $N + 1$ th dimension must be added to the input vectors. The original dimensions are interpreted as angles and the $N + 1$ th dimension represents the length of the vector, which is chosen the same for all inputs [Miikkulainen, 1987].

The weight change is proportional to the activity of the unit. This means that only units within the final focused response do change. The neighborhood size can be gradually reduced by increasing lateral inhibition over lateral excitation (through a global parameter) [Miikkulainen, 1987].

Measuring the similarity, neighborhood selection and weight change are all implemented with local computations in the biological model, without a need for an external supervisor.

6.2.3 Computational abstractions

The biologically inspired mechanisms above are computationally intensive, and the process is sensitive to the lateral weight parameters and the sigmoid parameters. Settling the activity is the bottleneck process. Fortunately, settling can be abstracted without changing the nature of the self-organizing process itself.

Euclidian distance is usually used as the similarity measure, so that the input vectors do not need to be normalized¹. The response is scaled between 0 and 1, with 1 indicating zero

¹The length of the vectors does not affect the Euclidian difference like it does the scalar product. For example, the scalar-product similarity of (1.0 1.0) and (1.0 1.0) is 2.0, while it is 0.5 for (0.5 0.5) and (0.5 0.5). The Euclidian difference is 0.0 for both.

distance:

$$\eta_{ij} = 1.0 - \frac{\|x - m_{ij}\|}{d_{max}}, \quad (6.4)$$

where x is the external input vector, m_{ij} is the weight vector of unit (i, j) and d_{max} is the maximum distance of two vectors in the input space (e.g. $\sqrt{2}$ in the 2-D unit square). The response can also be passed through a sigmoid function.

In the simplest version the similarity is computed only to determine which unit would responde maximally, and it is not necessary to compute the actual responses. An external supervisor finds the unit with the smallest distance, looks up the current neighborhood radius from a training schedule, and tells units within this radius of the maximally responding unit to adapt their input weights. No lateral connections nor settling need to be modeled to make this decision.

In some cases (e.g. in the trace feature maps of section 8.2 and in the lexicon, section 9.3.3), it is necessary to model the focusing of the response more accurately. This can be done by scaling the distance-based response within the neighborhood:

$$\eta_{ij} = \begin{cases} 1 - \frac{\|x - m_{ij}\| - d_{min}}{d_{max} - d_{min}} & \text{if } (i, j) \in N_c \\ 0 & \text{otherwise} \end{cases} \quad (6.5)$$

where m_{ij} is the unit's weight vector, x is the input vector, N_c is the neighborhood around the unit closest to the input vector, d_{min} is the smallest and d_{max} is the largest distance of x to a unit in the neighborhood. This formula generates a nice concentrated activity pattern around the maximally responding unit.

The weight changes within the neighborhood are proportional to the Euclidian difference. More specifically, the weight components of unit (i, j) in the map are changed according to

$$\Delta\mu_{ij,h} = \begin{cases} \alpha(t)[x_h - \mu_{ij,h}] & \text{if } (i, j) \in N_c(t) \\ 0 & \text{otherwise} \end{cases} \quad (6.6)$$

where $N_c(t)$ is the neighborhood around the maximally responding unit (shrinking with time). Note that it is not necessary to compute the actual response of the unit to adapt its weights, since this is taken into account in the Euclidian difference.

6.2.4 Example process

Self-organization of the weight vectors with the abstract implementation of section 6.2.3 is visualized in figure 6.3. In this simulation, the input data consisted of 2-D vectors uniformly distributed on the unit square. The input weight vectors of the 2-D feature map are initially uniformly distributed on the unit square. There was no dimensionality reduction in this case, so that the weight vectors could be plotted on the input space. During self-organization, they gradually become an ordered map of the input space. First a number of units are clustered together, and as the neighborhood size decreases, they gradually separate and cover the whole space. In the final map, each unit is responsible for an approximately equal area of the input space, except at the boundaries, where the map is slightly contracted.

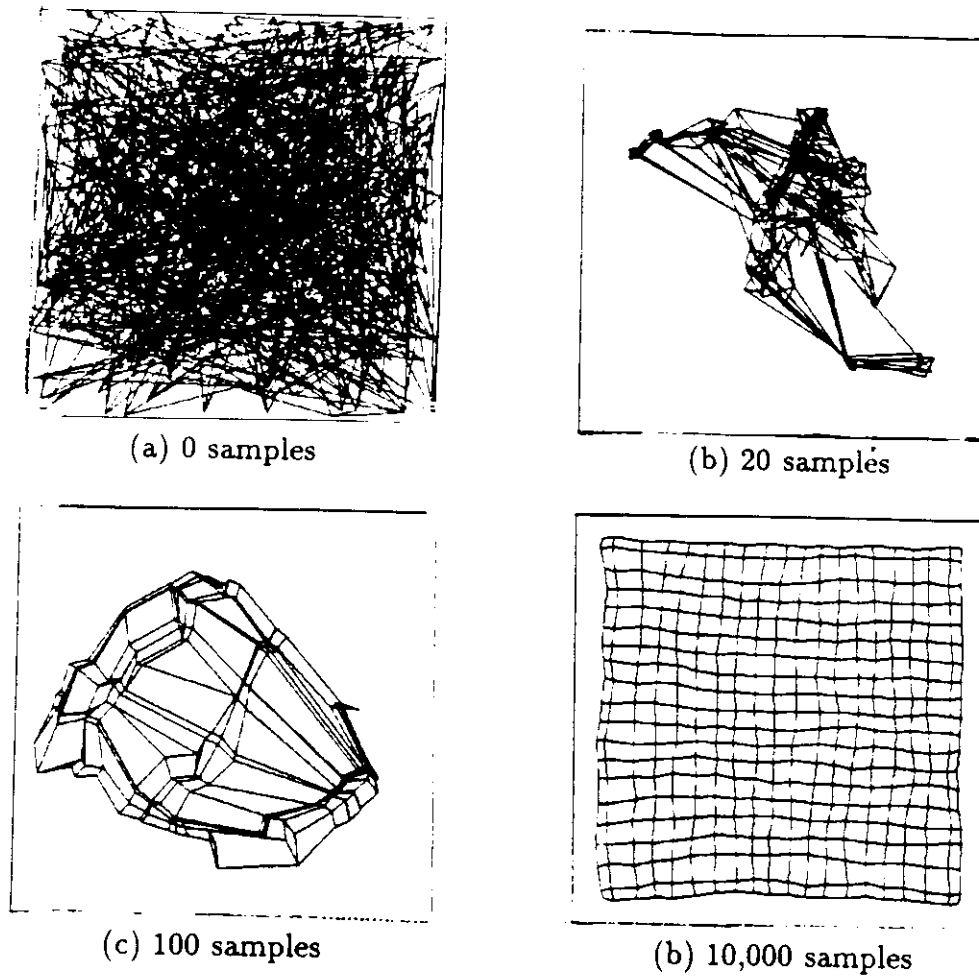


Figure 6.3: **Self-organizing the weight vectors.** The map consisted of 20×20 units in a 2-D array organization. The weight vector of each unit is displayed as a point on the unit square. Each of the weight vectors is connected with a line to the weight vectors of the four neighboring units. In other words, the vectors are located at each intersection and end point of lines, and the grid indicates the topological organization of the units. The figures display organization after 0, 20, 100, and 10,000 input vector presentations. The neighborhood size was decreased from 7 to 2 1000 presentations, and from 2 to 0 during the rest of the simulation. At the same time, α was decreased from 0.3 to 0.05, and then to 0.0.

6.3 Feature maps as memory models

Feature maps have several potentially useful properties for modeling memory. The key property is that the distributed representations of the input data are laid out spatially over the map, assigning different locations to different items. In other words, *both the distributed and localist features are combined in the representation*. It is possible to implement processing functions with associative connections between maps.

(1) The classification performed by a feature map is based on a large number of parameters (the weight components), making it very robust. Incomplete or somewhat noisy input items can usually be correctly recognized.

(2) Once an inexact input item is recognized, it is possible to recover its exact representation from that location categorically and exactly, *without blending*.

(3) The map is continuous, and contains intermediate units which do not stand for any particular input item, but represent combinations of items. In other words, in some cases it is *also possible to recover a blend* of two items.

(4) Several items can be active on the map at the same time, i.e. different alternatives can be represented in parallel. With associative connections between maps, many-to-many mappings are possible.

(5) Because items are stored in different parts of the map, *one-shot storage is possible*. Storing a new trace does not necessarily affect all other traces on the map.

(6) Differences between the most frequent input items are magnified spatially in the mapping, i.e. the variations of the most common event sequences or word meanings or lexical items are more finely discriminated.

(7) The self-organizing process requires no supervision and makes no assumptions of the content of the input data. The properties of the items which best distinguish between them are determined automatically, and may be very different for different kinds of items.

Chapter 7

Episodic memory organization: hierarchical feature maps

The main requirements for the episodic memory are (1) organization according to the similarities in the data, i.e. the taxonomy of script-based stories, and (2) being able to store items one-shot, i.e. with a single presentation and without knowledge of additional items to be stored. The reader can see that the feature maps as presented in the previous chapter form a promising starting point, but need to be augmented with additional mechanisms. This chapter concentrates on the organization of the memory. The storage and retrieval of traces is discussed in chapter 8.

The self-organizing process produces a representation of the input space where the input data is spread out spatially on e.g. a 2-dimensional sheet. The map represents most directly input spaces which are continuous and unstructured. However, script-based story representations are strongly hierarchical. Each script class consists of a number of tracks, and each track can be instantiated with different role bindings (figure 7.1). The 2-dimensional map tries to indicate the hierarchy through topological order [Kohonen, 1984]. The map is essentially divided into three main areas for the three different scripts, and each is divided into three subareas for the different tracks (figure 6.2). Knowing what the hierarchical taxonomy of the data is, it is easy to see that the spatial layout of the map reflects the taxonomy. However, as discussed in section 6.1, the boundaries of the continuous areas are not marked on the map, and it is hard to *extract* the taxonomy from the map alone.

Hierarchical feature maps [Miikkulainen, in press] can be used to make the hierarchical taxonomy explicit. This taxonomy provides a meaningful organization for the episodic memory. The hierarchy also cuts down the number of redundant system parameters (input weights) and concentrates the resources on the most salient aspects of the input data. The computationally intensive training is speeded up by effectively dividing the mapping task into hierarchical subgoals.

7.1 The hierarchical feature map architecture

A hierarchical feature map system is a pyramid of feature maps, which self-organizes to reflect the hierarchical taxonomy of the input data. The system can then be used to determine the hierarchical classification of an input item, and when used as organization for memory, assign a unique memory location for the item based on this classification.

The highest level of the hierarchy is first laid out on a single map by the normal self-organizing process (figure 7.2). A small map size forces the process to make only a gross, high-level classification. The units in this map stand for the highest level categories.

Each unit in the top-level map has another feature map beneath it, as does each unit in

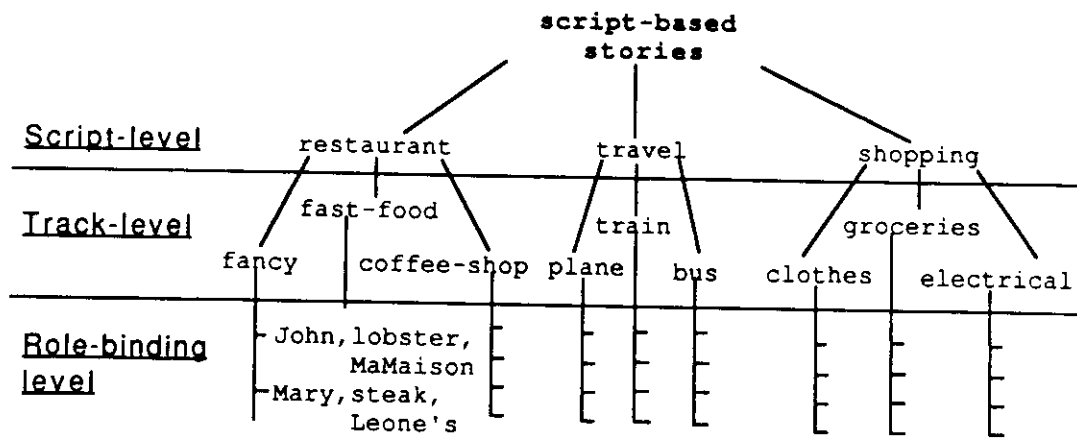


Figure 7.1: The hierarchy of script-based stories. Each script divides into tracks, and each track can be instantiated with different role bindings.

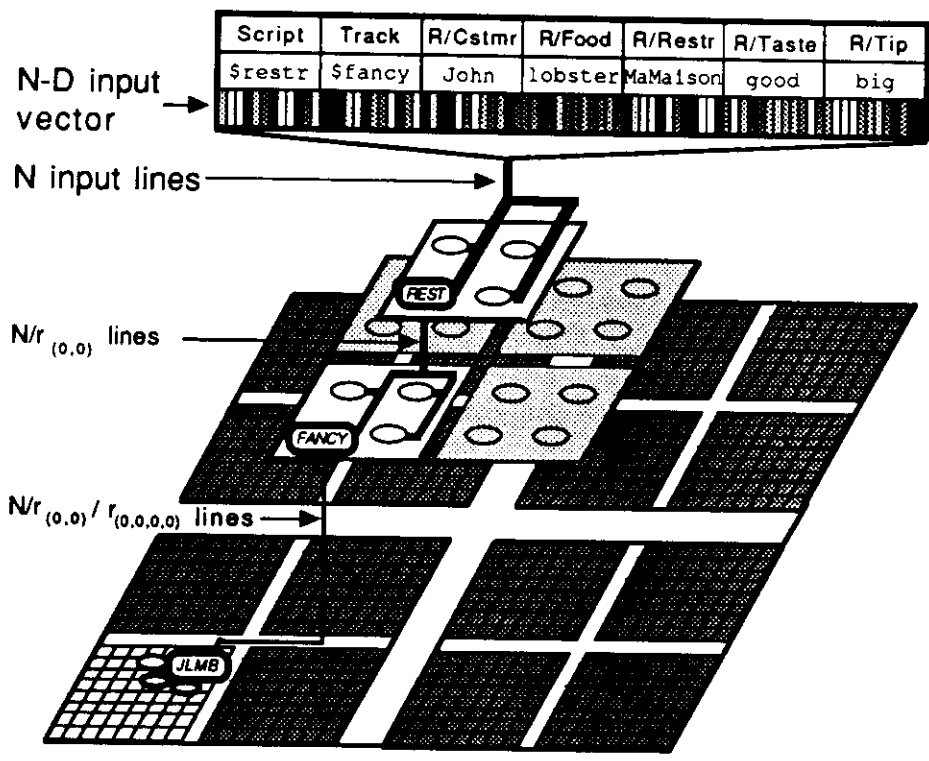


Figure 7.2: Hierarchical feature map architecture. Each unit keeps track of the variance on its input lines, and finds those whose variance is above a prespecified threshold value. It then passes the compressed input vector down to its submap for a more detailed mapping. The hierarchical classification of an input item is indicated by the maximally responding units at each level.

the second-level map. The system of maps thus forms a pyramid-like structure. A lower map in the structure receives as its input only those input items which belong to the category represented by its parent unit. In other words, the unit which "wins" the input item passes this item down to its submap. The lower map forms a subcategorization of these input items, mapping the differences within the category. The complete hierarchical classification of an input item is indicated by the maximally responding units at each level of the hierarchy.

For example, when the data consists of script-based stories (figure 7.1), the top-level map lays out the different scripts (figure 7.2). All restaurant-script stories are mapped on the unit labeled REST. The feature map beneath this unit receives only restaurant stories as its input, and self-organizes to represent variation within the restaurant stories, i.e. it lays out the different tracks of the restaurant script. Similarly, the bottom level maps out the different role bindings within the track. Each story is represented by three units in the map hierarchy: the image unit at the highest level indicates the script, the middle-level unit specifies the track, and the bottom-level unit indicates the role bindings of the story.

Input representations in a cognitive system often have some discrete structure, such as e.g. role specific assemblies [Hinton, 1981; McClelland and Kawamoto, 1986; Miikkulainen and Dyer, 1989a]. If the data is hierarchical, the items belonging to the same category have a number of components in common. These components can be removed from the input to the next-level map. The higher-level map acts as a filter, (1) choosing the relevant items for each submap (e.g. only restaurant stories for the restaurant-submap), and (2) compressing the representation of these items to the most relevant components before passing them on for a more detailed mapping.

Each unit has to determine independently which of its input lines are the most significant in distinguishing between items of the category. It has to find the *lines with the most variation* between the items it wins. These lines are different for different units. For example, a unit which stands for the fancy-restaurant track of the restaurant script wins only items which have \$fancy represented in the track assembly (figure 7.3). The values on the input lines of this assembly do not change from one input to another, and they can be removed from the input to the submap. On the other hand, the R/food may be different in different fancy-restaurant stories, and lines representing the IDs of the different foods have a lot of variation (figure 7.4). These lines are necessary to determine the role bindings of the track, and they are therefore passed on to the lowest-level map.

It is most efficient to start the self-organizing process at the highest level, and include a lower level only when the higher level has become ordered. Each unit keeps track of the variance in its input lines, and passes on lines whose variance is above a prespecified threshold value. As a result, the number of input lines at lower levels varies from map to map, i.e. the extent of the compression adapts to the data.

In practise, the variance only needs to be computed during the last epoch before including the lower level, because the higher level map is ordered and the mapping of inputs to units (and consequently, the variance) is stable. The variance information on each line is then reduced into a single bit, indicating whether or not the line is included in the compressed input vector the unit passes on to its submap (see function `choose_indices` in appendix C.2.1).

The variance threshold depends on the structure of the input data. In DISCERN, each

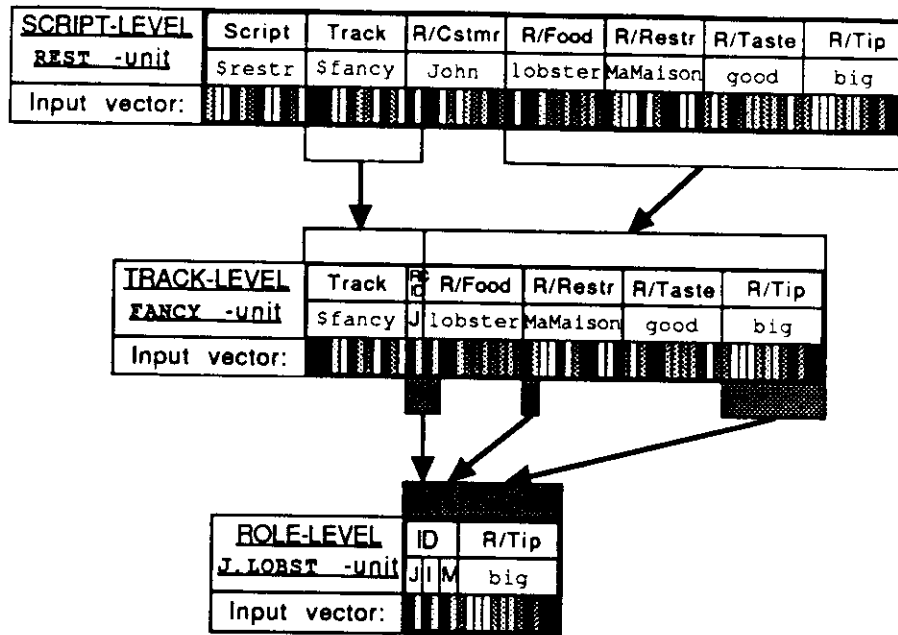


Figure 7.3: **Compression of the input lines.** All stories mapped on the REST-unit (figure 7.2) share the script `$restaurant` and the content part of R/customer, and these components can be dropped from further mapping. All inputs mapped on the FANCY-unit have `$fancy` as the track and `good` as R/taste, and share the same content parts in the R/food and R/restaurant roles. The input to the role level consists of only the IDs of R/customer, R/food and R/restaurant, and R/tip.

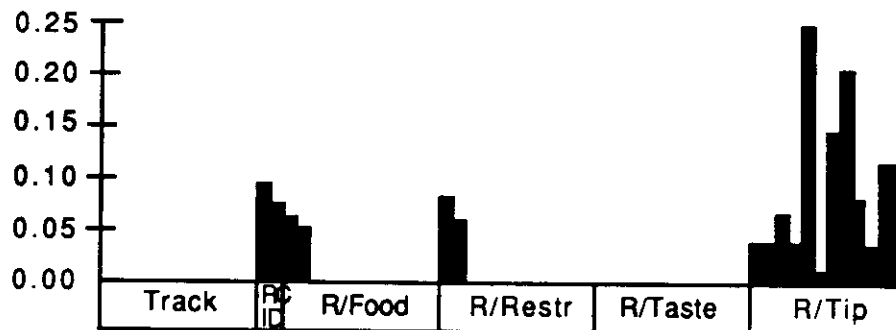


Figure 7.4: **Variance on the input lines of FANCY-unit.** The variance is calculated over all vectors mapped on this unit in one epoch. Most lines have zero variance, i.e. all vectors have the same values on these lines. Only the IDs and the lines representing R/tip vary between vectors, and these lines are passed on to the next level.

unit passes to its submap all input lines with nonzero variance. The decision is very robust, because the story representations are strongly structured, and the input components either vary a lot or not at all. Passing on all lines with nonzero variance guarantees maximum accuracy in the map representation, although the classification would be possible with more extensive compression also.

7.2 Hierarchical feature maps in DISCERN

7.2.1 Organizing episodic memory

The hierarchical map organization for the episodic memory can be formed simultaneously while training the FGREP modules, using the gradually developing representations. However, it is more efficient to train the FGREP modules first and use the final representations in organizing the episodic memory.

Complete story representations, i.e. targets for the story parser network, with randomly set IDs are used as input data. The hierarchical feature map system thus gets to see a sample of the space of possible IDs, and develops a mapping of the whole space of possible script-based stories. Key parts of the training code, simulation specification file and samples of training data are listed in appendix C.2.

A three-level pyramid of feature maps was used to form the script taxonomy (figure 7.5). The top level, a single 2×2 map, became ordered in 6 epochs. Restaurant, shopping and travel stories are mapped onto different corners of this map. Of the 84 input lines to the top level, each unit learned to remove the input lines in the script-slot and the content part of the customer/traveler slot, because the values on these lines are the same for all stories based on the same script. The TRAVEL-unit also learned to remove the lines in the last assembly and the SHOPPING-unit in the last two assemblies, because these assemblies are always zero (unused) in these scripts. Each unit passes on the rest of the input lines to its submap for a more detailed mapping, i.e. the REST unit passes 62 lines, the TRAVEL unit 50 lines and the SHOPPING unit 38 lines.

The second level consists of four 2×2 maps. In another 6 epochs, each of the maps categorized their compressed input vectors into different tracks, which were laid out on different corners. All these units passed on the 6 lines representing the IDs of the three open roles of the script, e.g. R/customer, R/food and R/restaurant. FANCY and COFFEE also passed on the 12 lines of the tip-role, which alternate between representing large or small, and COFFEE passed an additional 12 lines of the taste-role, which could be filled with good or bad. Similarly, the TRAIN unit passed on 12 lines of the distance-role, which sometimes represented large, sometimes small in the training data. These lines completely specify the role bindings of the story.

The lowest level contains sixteen maps, 8×8 units each. These maps lay out the space of the possible role bindings for each track. These maps have a different character than the higher level maps, because they are mapping a continuous space rather than items from discrete categories. They must be presented with a good statistical sample of the space, and the training takes longer. A rough organization took place in about 25 epochs, while an

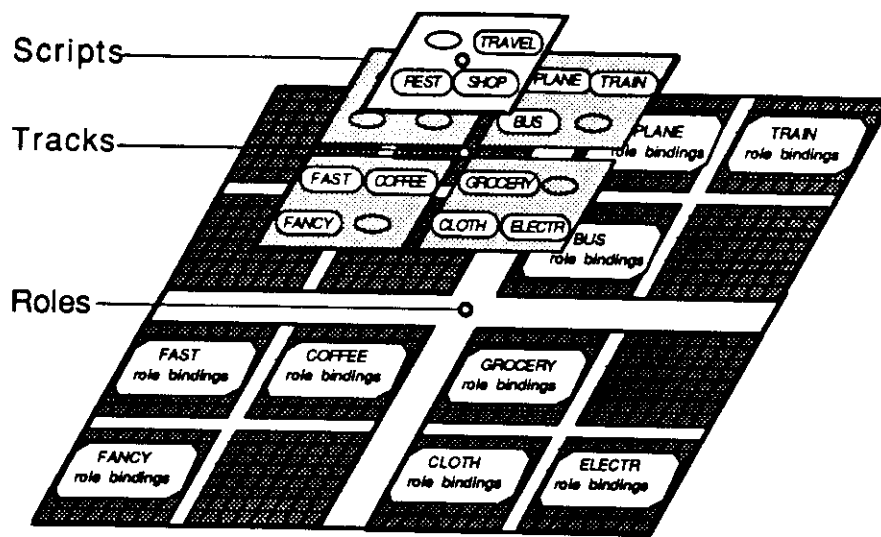


Figure 7.5: A hierarchical feature mapping of script-based stories. The highest level is a single 2×2 map, where the scripts are mapped onto different corners. The middle level consists of four 2×2 maps, which display the tracks of each script. The sixteen 8×8 maps at the lowest level map out the space of the possible role bindings for each track.

accurate mapping of the space was developed in about 1000 epochs.

The taxonomy can now be used to classify an input story representation as an instance of a particular script, track and role binding, and to assign a unique memory location for the story. For example, let's see how the story about John's visit to MaMaison is processed (figure 7.2). The top-level map receives the complete representation vector and maps it onto the unit labeled REST. This unit compresses the vector by removing the components whose values are the same in all restaurant stories (figure 7.3). The representation now consists of information which best distinguishes between the different restaurant stories. The REST-unit passes the compressed representation down to its submap, which classifies it as an instance of the fancy-restaurant track. Again, the FANCY-unit removes the components common to all fancy-restaurant stories, and passes the highly compressed vector to its submap. The representation is now limited to information about the role bindings, and it is mapped onto the unit representing customer=John, food=lobster, restaurant=MaMaison and taste=good.

7.2.2 Representation of the role binding space

During self-organization, each of the ID components is randomly distributed between 0 and 1. A map with 6 input lines therefore has to form a 2-D map of a uniformly distributed 6-dimensional unit hypercube. If the mapping is perfect, the weight vectors of the 64 units in the map are placed at the centers of disjoint $1/64$ th hypercubes. Each of the six dimensions is thus represented by $\sqrt[6]{64} = 2$ different values. These values are located in the centers of each dimension, i.e. at 0.75 and 0.25. This is the best possible representation of the 6-D space with 64 units. A map of this kind can separate up to four different ID patterns, and accurately represents the patterns 0.25,0.25, 0.25,0.75, 0.75,0.25 and 0.75,0.75. In other

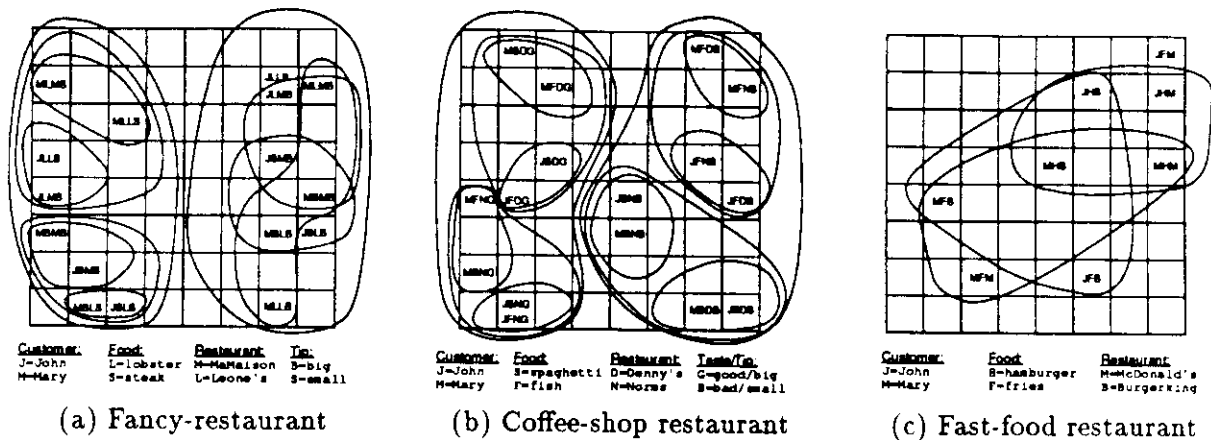


Figure 7.6: Role binding maps for the restaurant script.

words, the map can lay out a role-binding space with upto four instances of each prototype word.

The maps develop representations which are very close to the ideal. The weight values representing the IDs are very close to 0.75 and 0.25. Most of the deviations occur in the FANCY, COFFEE and TRAIN maps, which have to represent one or two additional roles (taste, tip, distance). These do not have continuous fillers like the roles with IDs; there are two possible fillers (good/bad, big/small, and the taste and tip correlate in the coffee-shop-restaurant stories (good food \Leftrightarrow big tip, bad food \Leftrightarrow small tip). Consequently, the additional roles expand the role binding space by a factor of two. The FANCY, COFFEE and TRAIN maps have to represent two unit-hypercubes in the 64 units. There are no longer enough units to represent each ID dimension with two values, and these maps cannot accurately represent 4 clones for each prototype. Some combinations of ID values are lost, and there are more midrange values.

In the test data for DISCERN, two clones were generated from each word prototype, based on ID patterns 0.75,0.25 and 0.25,0.75. Because only half of the space is used, the classification is very tolerant to error, and even the most crowded maps separate the resulting stories fairly well. Figures 7.6 to 7.8 show the role binding mappings of DISCERN test stories.

The maps have the impossible task of laying out the topological similarities of the 6-D vectors on 8×8 maps. A larger map would try to do this by developing curved and intertwined areas, but there is not enough space for that here. Interestingly, the maps develop two different strategies to approximate the topological order: (1) form connected areas for as many values as possible, splitting up the rest of them, or (2) form hierarchies of uniform areas.

The plane-travel map of figure 7.8a is a good example of the first strategy. There are connected areas for traveler=Mary, origin=LAX and destination=JFK, whereas stories with traveler=John, origin=DFW and destination=SFO are split up.

The ordering of coffee-restaurant (figure 7.6b), on the other hand, displays hierarchical ordering of role bindings. This map has two main regions according to taste/tip. The

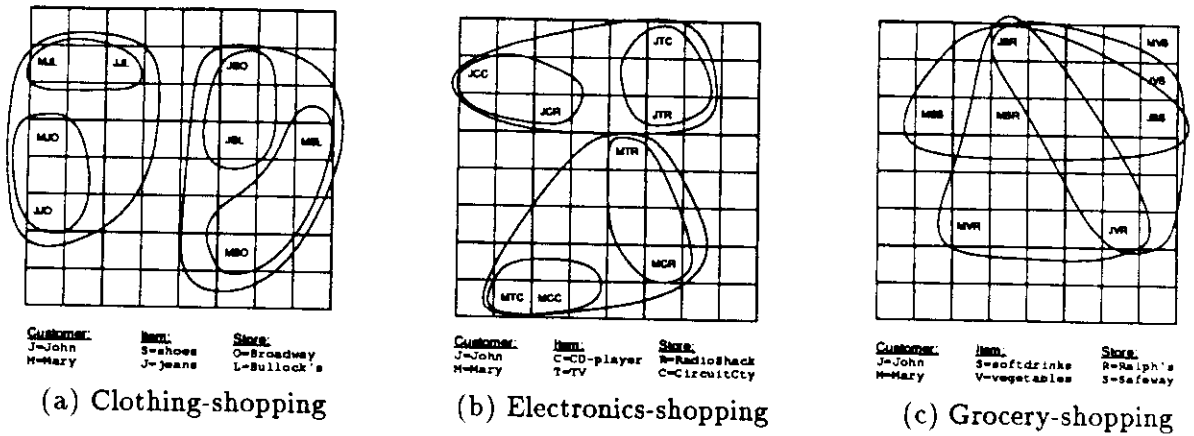


Figure 7.7: Role binding maps for the shopping script.

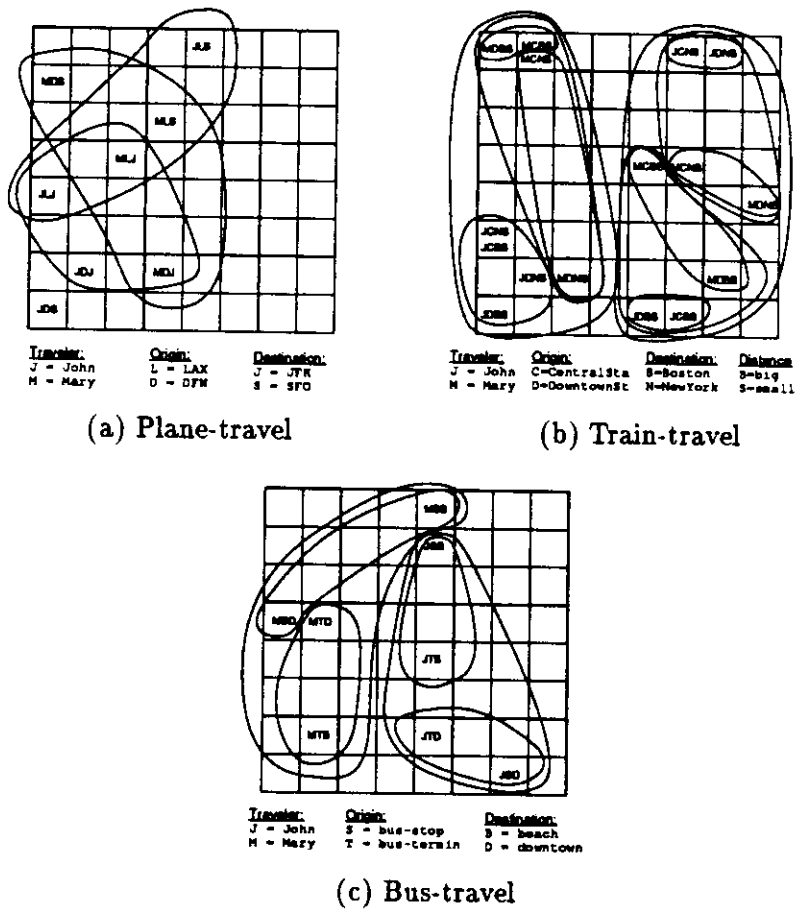


Figure 7.8: Role binding maps for the travel script.

good/big region divides further into two areas according to restaurant, and bad/small according to food. Each of these regions has two subareas where one of the remaining roles is constant.

The fancy-restaurant map employs both strategies. The tip=small subarea at left is hierarchically ordered, while the tip=big subarea at right has three uniform and three split up bindings.

It would be possible to organize the feature maps directly with the actual test data. With only two clones, this is an easier task than learning to represent the whole space from randomly assigned IDs, and the maps (especially FANCY, COFFEE and TRAIN) would develop more accurate representations for the particular ID patterns in the test data. The mapping would even be somewhat continuous, since the intermediate units develop intermediate values. However, the point of our training scheme is to demonstrate that the system can form a map of the whole role binding space without preconceptions about which stories are going to occur, and then use this space to represent the actual input stories. During performance, it is possible to continue adapting the maps to the input data, and the system performance will get better with more exposure to the actual data.

7.2.3 Retrieving representations

The representation of a story in the episodic memory consists of the image units at the script, track and role-binding levels. The full representation for the story can be recovered from the input weights of these units.

In the self-organizing process the input weight vectors become approximations of the input vectors [Kohonen, 1982a; Kohonen, 1984]. The weight vector at each unit represents an average of all stories mapped on that unit. The weights on constant input lines accurately represent the constant components, while the weights on the input lines with high variance represent averages. The high-variance lines were passed down in the hierarchy for a more accurate mapping, and the different values for these components are accurately represented at the lower levels of the hierarchy. A complete and accurate story representation can be retrieved by concatenating the accurate weights, i.e. weights on the compressed input lines, of the image units at all levels (figure 7.9).

The input weight vectors at different levels of the hierarchy represent *different levels of abstraction*. A top-level unit represents the skeleton of the script where the different tracks and role bindings have been averaged out. When the compressed top-level weights are combined with the track-level weights, the track becomes established but the role bindings are still unspecified. For example, in the place of the R/food (either lobster or steak) there is the representation of the general (fancy-food), which is an average of the two (figure 7.9). The bindings are finally established when the lowest-level weights are combined with the representation.

The accuracies of map representations for the test stories of DISCERN are shown in table 7.1 (the testing routines are part of the training program, listed in appendix C.2.1; samples of test data are included in appendix C.2.2). As can be seen from the table, the over-crowded role-binding maps FANCY, COFFEE and TRAIN provide slightly less accurate

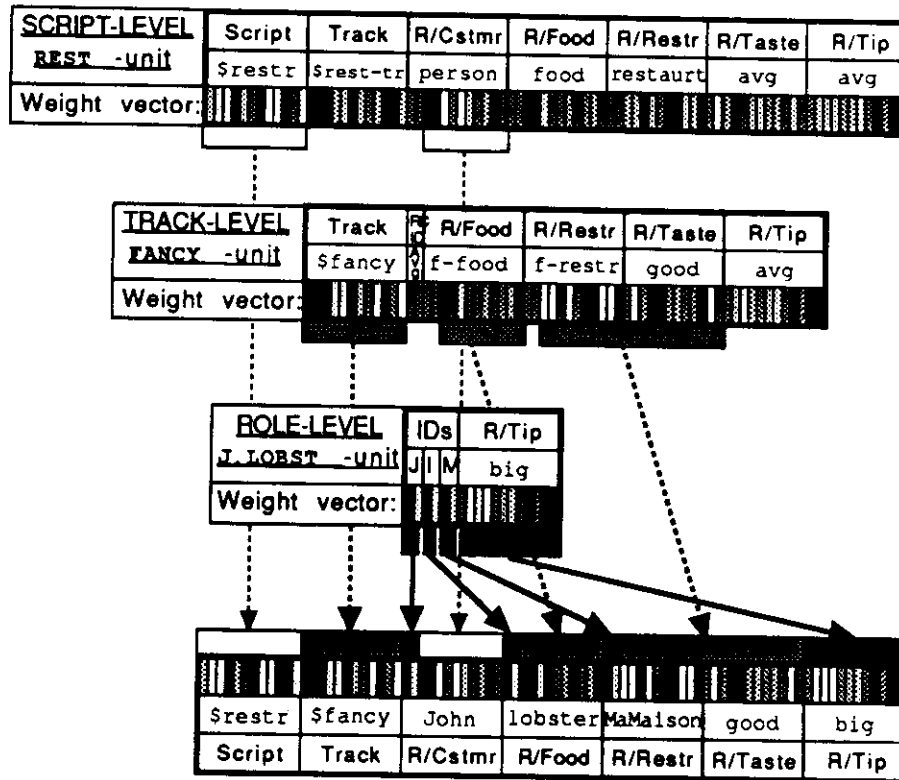


Figure 7.9: **Retrieving story representations.** Each unit knows which input lines it passes on to the next level. This information also serves to specify whether the weight on that line accurately represents a component, or whether it represents an average to be replaced by a weight from a lower level unit. The order of the components in the representation is specified at the top level. A complete story representation is retrieved by recursively replacing the weights on the lines which were passed on to the next level by the weights at the next level.

Map	All words	Instances	$E_i < 0.15$	E_{avg}
R-fancy	99	98	99	.0063
R-coffee	99	98	99	.0059
R-fast	100	100	99	.0056
S-cloth	100	100	100	.0036
S-electr	100	100	99	.0037
S-grocery	100	100	99	.0047
T-plane	100	100	99	.0056
T-train	98	96	98	.0070
T-bus	100	100	100	.0046
Average	99	99	99	.0055

Table 7.1: Accuracy of the hierarchical feature maps in representing DISCERN test stories. Two clones were created for each word prototype, with ID values 0.75,0.25 and 0.25,0.75.

representations, but not by very much.

The map representation is not only accurate, it also filters out noise and inaccuracies very efficiently. No matter how inaccurate the original input representation is, if it can be mapped correctly, the inaccuracies are automatically and almost completely removed in the weight representation. This error-correcting capability comes from the fact that there is only a limited number of categories, and the representation for each category is exact. The item representation is replaced by the correct representation of the category it belongs to.

Figure 7.10 depicts the accuracy of the map representation with noisy input data. The representation is very accurate, up to 30% noise level, and then deteriorates rapidly as classification breaks down. In other words, the episodic memory exhibits categorical perception. Input items with up to 30% noise are perceived and processed just the same as completely accurate ones.

The maps contain several units which are not images of any particular input item. These extra units are needed in the self-organizing process to develop a meaningful topological organization of the map. They constitute extra space that makes the mapping continuous. The weights on these units represent stories which are *combinations of the actual inputs* that the system has seen. In some cases they may be meaningful abstractions or generalizations, extracted from the similarities of the items around them in the map. Such a unit between two script units, for example, can represent uncertainty about the script class the incomplete input story belongs to. Some bindings can be safely filled in, and they are accurately represented in the weights of this unit. Others are uncertain, and the corresponding weights represent averages of the different possibilities. In many cases though, the unlabeled units do not represent anything useful. A combination of e.g. the airplane-travel script and the cake-baking script is not anything that occurs in real life. However, in a system consisting of a large number of scripts, these scripts would probably be mapped on different areas in the map. The map tends to maximize continuity, and most unlabeled units lie between scripts that are similar, and whose combinations are possible.

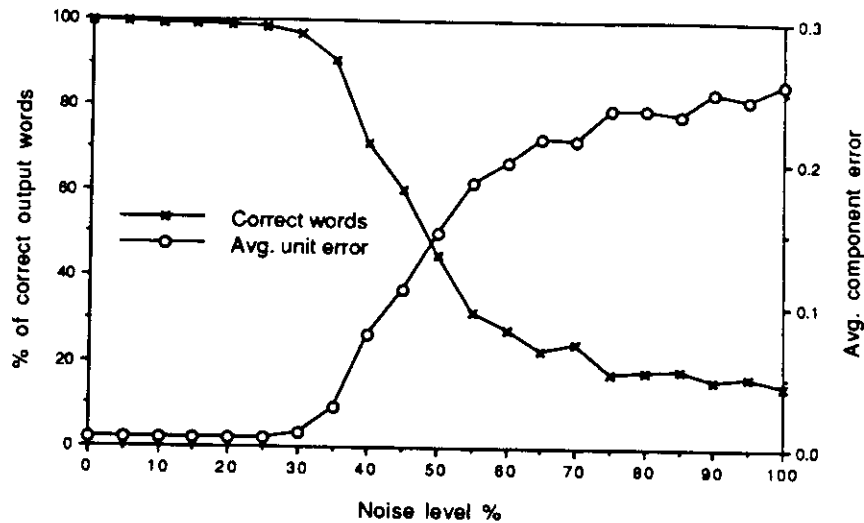


Figure 7.10: **Representation accuracy with noisy input data** The input vector components were obtained from $c_i = (1 - p)r_i + p * X$, where r_i is the story representation component, p is the noise level, and X is a random variable uniformly distributed within $[0,1]$.

7.3 Properties

7.3.1 Memory organization

Hierarchical feature maps have a number of properties which make them useful for memory organization:

(1) The maps visualize the data very nicely, with the hierarchy displaying the script taxonomy and the maps laying out the topology at each level.

(2) The organization is formed in an unsupervised manner, extracting it from the input examples.

(3) The most salient aspects of the input data are separated and most resources are concentrated on them. Within the script (or track), *only a subset of the representation is relevant to refining the classification further*. By focusing attention to this subset, the relative differences between the items are made greater, resulting in more robust self-organization and a more accurate final mapping. This is especially important with the kind of representation style we have chosen. A few representation components, i.e. the IDs which discriminate between the different instances, need to be stored and retrieved with extreme accuracy. The hierarchical feature maps employ abstractions to separate crucial information (instances, mapped at the bottom level) from information which is more widely distributed and where accuracy is not as critical (components common to the script and track, at higher levels).

(4) The classification is very robust, and usually correct even if the input vector is noisy or incomplete. The system can correctly classify a story representation as an instance of

the correct script and track even if a number of slots have inaccurate values. This is a consequence of the general redundancy of the system: the incomplete input is still closest to the correct script and track, and is classified accordingly.

(5) Self-organizing a hierarchy of small maps instead of a single large one means dividing the classification task into hierarchical subgoals, which is an efficient way to reduce complexity.

As was pointed out in [Schank and Abelson, 1977], people seem to make scripts out of all regularities in everyday experiences, and in several levels of complexity, abstraction and extent in time. The hierarchical feature map system can be seen as a general model of episodic memory, similar in spirit to the symbolic model of [Kolodner, 1984] (see section 12.5). Incoming experiences are classified in terms of their similarities to previous experiences (higher-level maps), and stored in the memory in terms of their differences (bottom-level maps). In doing so, the structure of the memory also changes, and affects future classification. The structure is extracted autonomically from examples, and reflects the moving statistical regularities in the data.

The hierarchical feature map model assumes that the whole space of possible episodes is represented in the hierarchy. The system has no mechanism for representing episodes which do not fit the memory organization. Novel episodes cannot be stored correctly. Interestingly, similar behaviour has been observed in hippocampal amnesia [Halgren, personal communication]. Certain patients with hippocampal damage cannot store memories of novel events, but they can create new traces of familiar events. A possible explanation is that two separate memory encoding processes exist. Familiar episodes are coded primarily by the cortex, whereas novel episodes require hippocampal activity. The hierarchical feature map structure can be seen as a model of cortical memory, making no claims about the processes involved in storing novel events.

Since the relevant roles are determined separately for each script, common roles end up being represented multiple times. Each script in our data has a role for the main actor, but this role has to be mapped separately on every map at the lowest level. Half of each role map is dedicated to stories with John as the main actor, the other half with Mary (figure 7.5). Retrieving e.g. all stories with John as the main actor becomes a fairly complex process (see section 8.3.3). Each map has to be scanned (in turn or in parallel) to see if appropriate traces exist. Interestingly, this appears to be a hard task for humans also. It seems that single role bindings are not good indices to episodic memory, but require a search process.

There are certain limitations to the hierarchical feature map approach. The resolution of the memory is defined by the number of units in the bottom level maps, and a combinatorial number of units are needed to represent the role bindings. These limitations and possible ways to overcome them in a more advanced architecture are discussed in section 13.4.1.

7.3.2 Self-organization

Self-organization in hierarchical feature maps implements hierarchical subgoaling [Korf, 1987]. The process begins at the highest level map, which divides the input space into a small number of high-level categories. After the high-level mapping has become established,

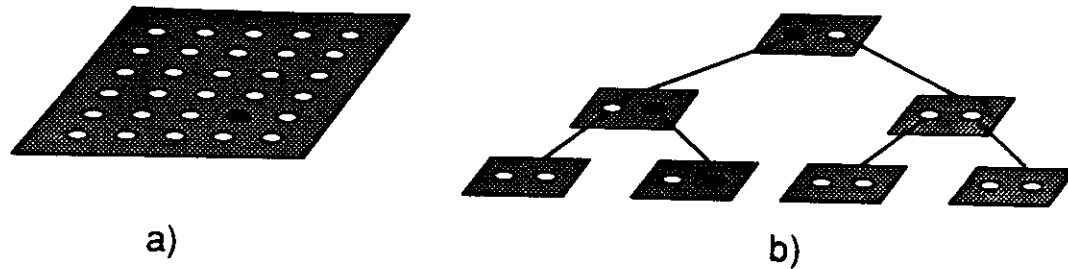


Figure 7.11: **A continuum of classification systems.** The hierarchical feature maps provide classification systems from a flat feature map to statistically balanced divisive clustering tree.

each subcategory is organized independently of others. This is a much easier task than organizing the whole space, because there are fewer dimensions of variation between the items in each category. For this reason, hierarchical self-organization is a very efficient way to reduce complexity.

Technically, the self-organization on a single-level map also employs hierarchical subgoal-ing. The initial neighborhoods cover almost the entire map [Miikkulainen, 1987]. This has the effect of dividing the space into a few large categories within the single map. The neighborhoods are then gradually decreased, allowing the map to make finer distinctions within categories, which corresponds to organizing maps lower in the hierarchy.

While forming the finer distinctions, the process on a single map also has to form and maintain the topological layout of the hierarchical categories (figure 6.2). For this reason, the neighborhood size must be decreased gradually, and the self-organization of each category interferes with the self-organization of the neighboring categories. The units near the category boundaries are modified towards the neighboring categories also, which slows down organization within the category.

The savings in the hierarchical feature map system come from isolating the self-organization of subcategories from the influence of other categories. The topology of the high-level categories is first laid on the top-level map, and the subcategories are then independently organized on separate maps. The subprocesses are free from maintaining the higher-level organization, and there is no lengthy decrease in the neighborhood size and no interference from other categories. In effect, in the single-level self-organization the topological map is formed in a single continuous process, whereas the *hierarchical self-organization proceeds in discrete phases*. The computational effort is saved in transitions between phases.

The hierarchical feature maps provide a continuum of classification systems, depending on the phasing of the process (figure 7.11). At one extreme there is the single-level feature map, which produces a single topological layout of the input space. At the other extreme there is a hierarchy of two-unit maps, which forms a statistically balanced divisive clustering tree of the input items: the inputs are first divided into two equally large classes, each class is further divided into two parts etc.

The hierarchical architecture makes most sense when the input data is strongly hierarchical, and the architecture of the system matches this hierarchy. In the extreme case of a uniform distribution of independent continuous variables there is little to be gained by using

hierarchies. If enough is known about the structure of the input space, *a hierarchical feature map architecture can be chosen, which self-organizes to directly represent the hierarchical semantics of the input space.*

In a single-level map all units need to receive complete representations of all input items (figure 6.1). Modifying the weights is costly, because the initial neighborhoods cover almost the entire map. In a hierarchical map system, the maps at lower levels receive only small subsets of the original input lines, and the neighborhoods are always small because the maps are small at all levels (figure 7.2). This reduces the training time considerably for serial implementations.

The story representation data has a lot of structure, and the reduction of the required input connections and the speed-up of the self-organization is quite dramatic. There were $4 \times 84 = 336$ connections to the highest level, $4 \times (62 + 50 + 38) = 600$ to the second, and $64 \times [(18 + 30 + 6) + (6 + 6 + 6) + (6 + 18 + 6)] = 6,528$ to the bottom level, i.e. a total of 7,464 input connections. The hierarchical organization was complete in only 35 epochs, which took about 3 minutes on an HP 9000/350 workstation.

A comparable single-level mapping (figure 6.2) with 576 units ($= 9 \times 64$, i.e. the same as in the active bottom-level maps) with the same training and testing data required 48,384 input connections, 300 training epochs and 14 hours to organize (simulation specifications are listed in appendix E.3.2). In this serial simulation, two orders of magnitude speed-up in time results from the reduction in the input connections and smaller neighborhoods. Another order of magnitude comes from the hierarchical subgoaling, and would apply to fully parallel implementations as well.

The outcome of the self-organizing process is somewhat sensitive to the system parameters. To obtain the three-level classification into scripts, tracks and role bindings the maps must have approximately the right size and each input story must have approximately the same frequency in the input data. If, for instance, the highest-level map is too large, or one of the script classes is far more frequent than the others, the different tracks may get separated already at the highest-level map. Even if the parameter settings are not ideal, the script recognition works the same. *The system uses whatever architecture is given to establish the best classification of the input data.* The configuration parameters may need to be experimentally adjusted to achieve the desired semantics for the levels of the hierarchy, but the function of the classification system is fairly robust.

On the other hand, the classification is purely statistical, and there is only one taxonomy that can be formed by a given architecture for given data. There is no way to include semantic information to guide the process, as in the supervised learning approach. Alternative indexing schemes, as e.g. stories organized according to the main actor, are not possible. If another organization is desired, the relevant information must be injected into the input vectors in a way that makes the desired organization the most descriptive.

Chapter 8

Episodic memory storage and retrieval: Trace feature maps

The hierarchical feature map structure provides the organization for the memory. The maps lay out the whole space of the possible input items on the available hardware, i.e. they assign a unique memory representation for each item. To use the map system as episodic memory, it must be possible to create traces of the items that the system has seen.

The memory representation of a story consists of the image units at the script, track and role-binding levels. However, the role binding unit alone stands for a unique story, and a trace needs to be created only at the role-binding unit. For this reason, the bottom level maps in the map hierarchy employ a special mechanism: they are trace feature maps.

An ordinary feature map is a classifier, mapping an input vector onto a location on the map. A trace feature map, in addition, creates a memory trace on that location. The map remembers that at some point it received an input item that was classified there. The traces can be stored one at a time, as stories are read in, and retrieved with a partial cue.

In this chapter, an ideal version of the trace feature map mechanism is first presented and its properties are illustrated and analyzed using uniformly distributed 2-dimensional input data. The code for this implementation is included in appendix E.5. The specific implementation of trace feature maps as part of the episodic memory in DISCERN is then discussed in section 8.2, with the code in appendix D.1.6. The trace maps are combined with the map hierarchy in the end of the chapter, where the storage and retrieval processes of the complete episodic memory model are described.

8.1 Trace feature maps

8.1.1 Initial set-up

In the biological model of self-organization, lateral connections between units are responsible for neighborhood selection (see section 6.2.2). In the trace feature map model, after the map has become ordered, *these same lateral connections are used to store the memory traces.*

The output η_{ij} of unit (i, j) on a feature map with lateral connections can be defined as:

$$\eta_{ij}(t) = \sigma \left[(1 - \theta) \left(1 - \frac{\|x - m_{ij}\|}{d_{max}} \right) + \theta \sum_{k,l} \gamma_{kl,ij} \eta_{kl}(t-1) \right]. \quad (8.1)$$

where x is the external input vector, m_{ij} is the unit's weight vector, d_{max} is the maximum distance of two vectors in the input space ($\sqrt{2}$ in the 2-D unit square), $\gamma_{kl,ij}$ is the lateral connection weight on the connection from unit (k, l) to unit (i, j) , and $\eta_{kl}(t-1)$ is the

output of unit (k, l) at the previous time step (see function `unit_resp` in appendix E.5.1). The function σ is the familiar sigmoid function,

$$\sigma(z) = \frac{1}{1 + e^{\alpha(\beta - z)}} \quad (8.2)$$

where the parameter α determines the slope of the sigmoid and β its displacement from the origin (function `sigmoid` in appendix E.5.1). In other words, each unit computes a weighted sum of its lateral activity, adds the activity resulting from the external activation¹, and develops an output activity which is a sigmoid of the sum. The parameter θ determines the balance of external and lateral activation, and it is used to separate the storage and retrieval on the trace feature map.

During self-organization, $\theta > 0$, and the lateral connections implement lateral inhibition. The inhibition vs. excitation ratio is gradually increased, decreasing the weight change neighborhoods as required for self-organization (section 6.2.2). We also assume that the θ parameter is gradually decreasing so that the response becomes more sensitive to the external input as the map becomes more ordered. At the limit, the weight vectors form a topological map of the input space, $\theta = 0$ and the lateral connections are all inhibitory. This is the ideal initial configuration for the trace feature maps.

However, the trace feature map mechanism is independent on the self-organizing process. The map can be formed using lateral connections (as described above), or with a computational abstraction (as in DISCERN) or by simply assigning the right weights to the units to begin with (as is done below).

8.1.2 Storage mechanism

Let us assume that we have developed an ordered 2-D feature map of a uniform distribution in the unit square with the above process. The input vectors in this case are 2-dimensional, with each component uniformly distributed within $[0, 1]$. The weight vectors of the ordered map form a regular grid on the unit square, where each unit is responsible for an equal area of the input space (except the units near the boundary, which are shifted slightly towards the center in the self-organizing process) (figure 8.1).

Let us further assume that the lateral connections of the map are all inhibitory with $\gamma_{kl,ij} = \gamma_I$ (a constant). This means that the map is blank, contains no traces. During storage, $\theta = 0$ so that the response of the map depends only on the external input activity.

When this map is presented with an input vector, its response is a localized symmetric “bubble” around the unit whose weight vector is closest to the input vector (figure 8.1). The width and the intensity of the bubble depend on the sigmoid parameters.

A trace is created by modifying the lateral connections of the active units². For each unit in the bubble, a connection to a unit with a higher activity is made excitatory, while

¹In computing the external response, the Euclidian distance abstraction is used for simplicity. A biologically more plausible but computationally cumbersome alternative was discussed in section 6.2.2.

²Since $\sigma(z) > 0$, a unit is considered active if its output $\eta > \eta_a$, where η_a is a suitable threshold value, e.g. 0.1

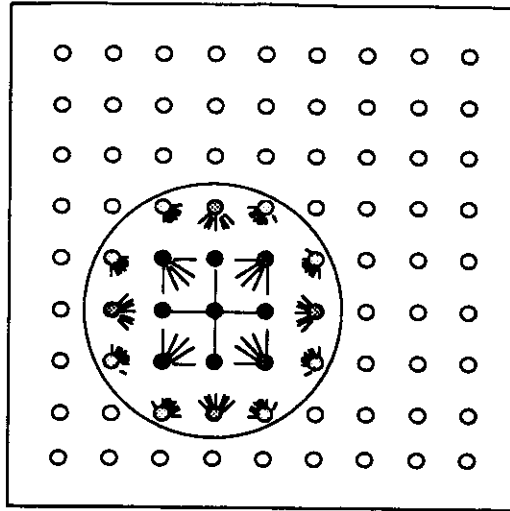


Figure 8.1: **Creating a trace.** The map lays out a uniform distribution in the unit square. The coordinates of each unit correspond to the two components of its weight vector. The gray-scale coding indicates the response of the units to the external input $[0.4, 0.4]$, with $\theta = 0.0$, $\alpha = 10.0$, $\beta = 1.4$. The line segments indicate excitatory lateral weights emanating from each unit. The lateral weight parameters were $\gamma_E = 2.0$, $\gamma_I = -0.5$. Only the excitatory lateral connections are shown.

a connection to a unit with a lower activity is made inhibitory, both proportional to the activity level of the source unit:

$$\gamma_{ij,kl} = \begin{cases} \gamma_E \eta_{ij} & \text{if } \eta_{ij} < \eta_{kl} \text{ or } (i, j) \equiv (k, l) \\ \gamma_I \eta_{ij} & \text{if } \eta_{ij} \geq \eta_{kl} \end{cases} \quad (8.3)$$

where $\gamma_{ij,kl}$ is the connection from unit (i, j) to unit (k, l) and $\gamma_I < 0$ and $\gamma_E > 0$ are the inhibitory and excitatory connection strengths (see function `modify_weights` in appendix E.5.1). The units within the response are now “pointing” towards the unit with the highest activity in the bubble (figure 8.1).

The trace is created *in a single presentation*, and it is not necessary to know what is already in the memory and what additional items need to be stored later. This is not possible with most neural network models of associative memory because of crosstalk between traces (see section 12.6.5). In the trace feature map approach, different areas of the memory hardware are allocated to different items, largely eliminating the crosstalk.

8.1.3 Retrieval mechanism

During retrieval, the θ parameter in equation 8.1 is made positive, e.g. 0.5. The response of the map now depends both on the external input vector and the lateral connections, which code the stored traces.

A stored vector is retrieved by presenting the map with an approximation of the vector. The initial response is again a localized activity pattern (figure 8.2). Because the map is

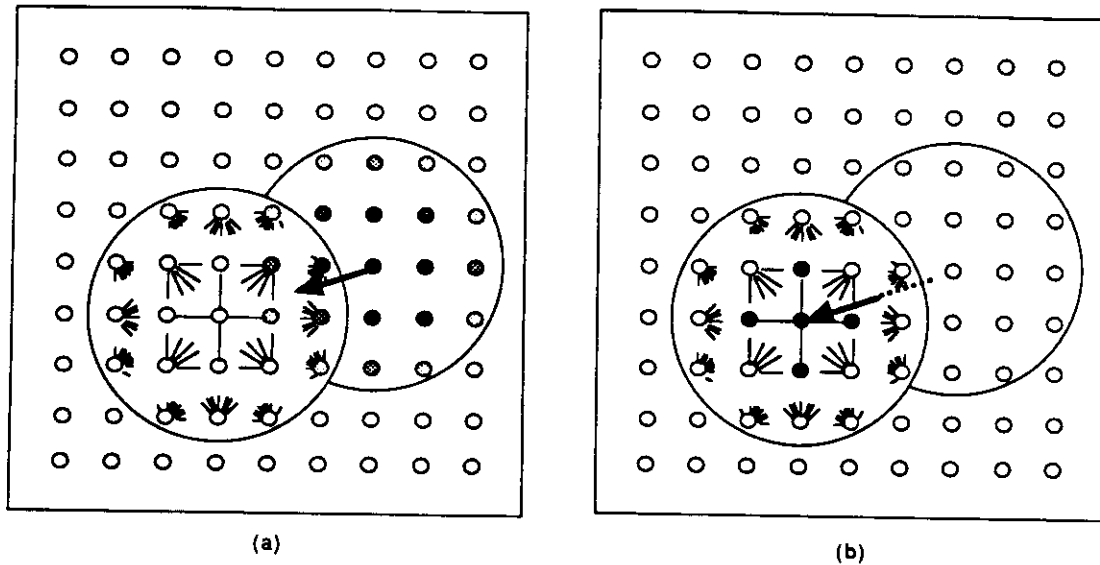


Figure 8.2: **Retrieval from a trace feature map with a partial cue.** The initial response to vector $[0.7, 0.5]$ (a) is pulled to the center of the trace $[0.4, 0.4]$ (b) by the lateral connections.

topological, the center of this pattern is somewhere near the target trace. If the cue vector is close enough to the target, the initial response overlaps with the trace. In the next few iterations of equation 8.1, the excitatory lateral connections within the trace pull the activity to the center of the trace. With proper setting of the parameters, the activity settles around the center of the trace, and external input weights of the unit with the highest activity give the stored vector.

If the cue vector is too far from the target, the initial response does not overlap with the trace. The lateral connections of the units within the initial response are all inhibitory, and in the next step, all activity is turned off. The next step then again is the initial response pattern, and the activity oscillates between nonactivity and the initial response. This oscillation indicates that there is no appropriate trace in the memory.

Notice that the retrieval process makes no distinction between incomplete, noisy and partly incorrect cues. Any pattern can be used to cue the memory, and if there is a trace close enough to the cue, it will be returned. As a result, incorrect cues are automatically corrected.

8.1.4 Memory effects and capacity

The trace map exhibits interesting memory effects which result from interactions between traces. When two traces are stored close to each other, the later one steals units from the older trace (figure 8.3). When a cue is mapped in the general neighborhood of the two traces, the newer one is likely to receive more lateral activation, which allows it to inhibit the older trace and eventually turn it off. This results in a recency effect: if several similar items are stored on the same map, the later traces are more likely to be retrieved than earlier ones.

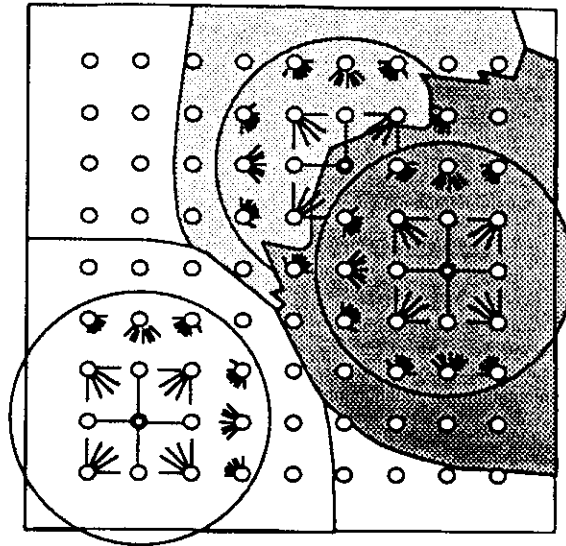


Figure 8.3: **Interaction of nearby traces.** The trace on the right has partially obscured an earlier trace. Traces are shown with circles, as before. The three shaded areas indicate the attractor basins for each trace. The areas were determined by presenting input vectors from a 200×200 grid covering the unit square, and noting which (if any) trace was retrieved. The weight space and the input space are superimposed in this figure. For example, any vector from the bottom left area of the input space will retrieve the bottom left trace. The same parameter settings were used as in the figures 8.1 and 8.2.

When the memory becomes overloaded, the older traces become increasingly hard to retrieve, and eventually they are completely replaced by the newer ones. However, this process is selective in that traces in a sparse area of the map are not affected, no matter how old they are. In other words, unique items are preserved.

The map in figure 8.3 shows the attractor basins for each of the three traces stored on the map. Two of the traces are located in the same area, and the later of them has partially obscured the older one. The third trace is located in an area by itself, and its basin covers a large area of the map.

It is difficult to characterize the memory capacity of trace feature maps. The memory effects are an important part of the model, but they make capacity analysis very hard. One obvious factor is the area of the trace. The smaller the basins, the more traces will fit in the map without obscuring each other. On the other hand, more accurate cues are also required to retrieve them.

Rather than attempting to provide an estimate on how many traces can be stored in a particular trace map, it makes more sense to outline the behaviour of the memory as it becomes increasingly loaded with traces. Figure 8.4 depicts some such performance descriptors as the function of the number of traces stored in the memory.

There is a 15% chance that a random vector will retrieve a single trace, i.e. on the average, the basin of the trace covers 15% of the map. As more traces are stored on the same map, the basin becomes smaller. With six traces, each one has only 7.5% chance of

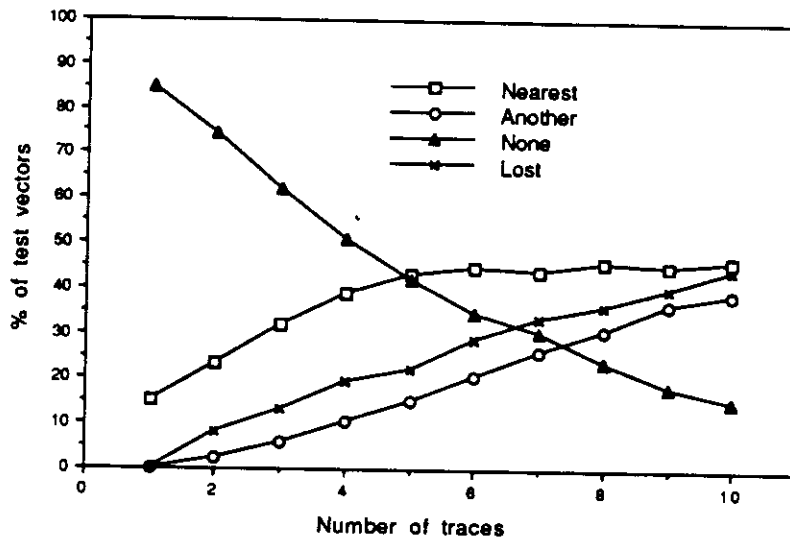


Figure 8.4: **Performance descriptors with increasing number of traces.** The map consisted of 9×9 units. The parameters used in this simulation were $\theta = 0.5$, $\alpha = 15.0$, $\beta = 1.4$, $\gamma_E = 0.5$, $\gamma_I = -0.075$. As a result, the radius of the trace was about 0.28 (as also in figure 8.3). The traces were uniformly distributed on the unit square. Each percentage represents an average of 100 trials. The test vectors were laid out on a 20×20 grid, which uniformly covered the unit square. In each trial, the test vectors were presented to the map one at a time, and depending on which trace they retrieved (if any), they were classified as "Nearest", "Another" or "None". The fourth possibility, settling onto a unit which was not a center of any stored trace, only occurred in about 0.1% of the trials, and was not plotted in the figure. If a trace was not retrieved at all during the trial, it was counted as a "Lost" trace. Typical standard deviations for each descriptor were: Nearest 12, Another 6, None 10, Lost 14.

being retrieved with a vector that is closest to them. The percentage of "Nearest" levels off, indicating that any additional trace will only steal units from the older ones.

Even when there are only two traces on the map, there is an 8% chance that the later one of them completely wipes out the earlier one. The percentage of lost traces grows approximately linearly as more of them are stored on the same map. With eight traces on the same map, 1/3 of them are inaccessible. This is surprisingly little, since a single trace not cut by the boundaries covers about 1/4 of the map.

The recency effect can be clearly seen from the graphs. The number of test vectors which will retrieve a later trace instead of the nearest one on the map grows approximately linearly with the number of traces stored. As the memory becomes overloaded, the oldest traces are gradually replaced by newer ones. It is necessary to specify more and more accurate cues to retrieve old traces, and eventually they become inaccessible altogether. With ten traces on the same map, a random cue is as likely to retrieve a later trace than the nearest one. At that point, 44% of them have become inaccessible altogether.

The conclusion is that the trace feature map memory degrades gracefully when it is overloaded. On the other hand, memory effects are possible even under a very light load. Statistically the behaviour is very predictable, but the different cases vary a lot. The standard deviations of the above percentages were typically greater than 10.

8.1.5 Tuning the parameters

The operation of trace feature maps is highly sensitive to the sigmoid and lateral weight parameters. The settling behaviour, extent and accuracy of the final stable pattern, distinction of retrieval and failure, and the maximum distance for retrieval can be controlled with these parameters.

Ideally, the settling should be continuous and gradual, where each successive pattern is closer to the final stable pattern. In many cases it resembles damped oscillation, i.e. the activity alternates between two patterns which both converge to the same final pattern. In one of these patterns the lateral connections dominate, the other one reflects the external input activity. The oscillations occur when the lateral weights γ_E and γ_I are large. The oscillations can be reduced by reducing absolute values of both (or the θ parameter), and also by reducing the ratio $\frac{\gamma_I}{\gamma_E}$.

The extent of the final activity bubble depends on several parameters. The steeper the slope of the sigmoid, the more it is displaced towards infinity, and the higher the inhibition, the smaller the activity pattern. More focused responses can thus be produced by increasing α and β , or by increasing $\frac{\gamma_I}{\gamma_E}$.

The center of the activity can be most reliably located when the bubble is gently sloping, i.e. α is small. High values of α tend to saturate several units to 1.0.

The distinction between successful and unsuccessful retrieval is less simple in reality than has been indicated so far. Ideally we would want the system to always settle to a stable pattern with a highly active center, or else oscillate between the initial response and zero activity. However, the sigmoid never yields strictly 0 activity, so the oscillation actually

occurs between a pattern close to initial, and an almost zero pattern. Also, sometimes the system does not settle to a stable solution, but oscillates between two nonzero patterns, both of which indicate the same center. It is necessary to use a threshold to decide whether such oscillations should be interpreted as retrieval or failure. If the lower of the highest activities in the alternating patterns is greater than the threshold, retrieve, otherwise indicate failure.

The two cases can be made quite distinct by increasing the absolute values of the lateral weights, and the system never has to depend very much on the setting of the threshold. In our simulations, the failures remained below 0.1, while the successes were all greater than 0.9 (see sections 10.2 to 10.4). If it is necessary to control how distant cues will cause retrieval, the distinction can be made gradual. In this case, the threshold value will directly control the distinction of success/failure.

As the reader can see, optimal behaviour would require contradicting parameter values. Usually it is still possible to find compromise values that produce satisfactory performance, especially if the traces can be fairly large, and retrieval from oscillating patterns is acceptable.

8.2 Trace feature maps in DISCERN

In DISCERN, the trace feature maps have to operate in less than ideal conditions. The role binding maps lay out high-dimensional spaces on a small number of units, and the maps have more structure and less continuity than in the ideal case above. Similar vectors may sometimes be mapped on different areas of the map. Somewhat more elaborate mechanisms are necessary to guarantee robust operation in these conditions (see code in appendix D.1.6).

8.2.1 Storage

The initial response of a role binding map is usually not a regular activity bubble, but instead reflects the structure of continuous subareas of the map. To create a well-localized trace, it is necessary to focus this response around the most active area on the map. This can be done by lateral inhibition. Instead of $\alpha = 0$ and all lateral connections inhibitory as before, we now have $\alpha > 0$ (but small) and weak lateral inhibition, and response settles through equation 8.1. With appropriate values for α and the lateral inhibition, the final activity pattern still reflects the maximum of the external activation, but it is more focused.

Having established a plausible process for focusing the response, we can replace it with computationally more efficient implementation which produces the same effect. In the storage phase of trace feature maps in DISCERN, the response of the map to an external input is given by

$$\eta_{ij} = \begin{cases} 1 - \frac{\|x - m_{ij}\| - d_{min}}{d_{max} - d_{min}} & \text{if } (i, j) \in N_c \\ 0 & \text{otherwise} \end{cases} \quad (8.4)$$

where m_{ij} is the unit's weight vector, x is the input vector, N_c is the neighborhood around the unit closest to the input vector, d_{min} is the smallest and d_{max} is the largest distance of x to a unit in the neighborhood (see function `store_trace` in appendix E.5.1). In other

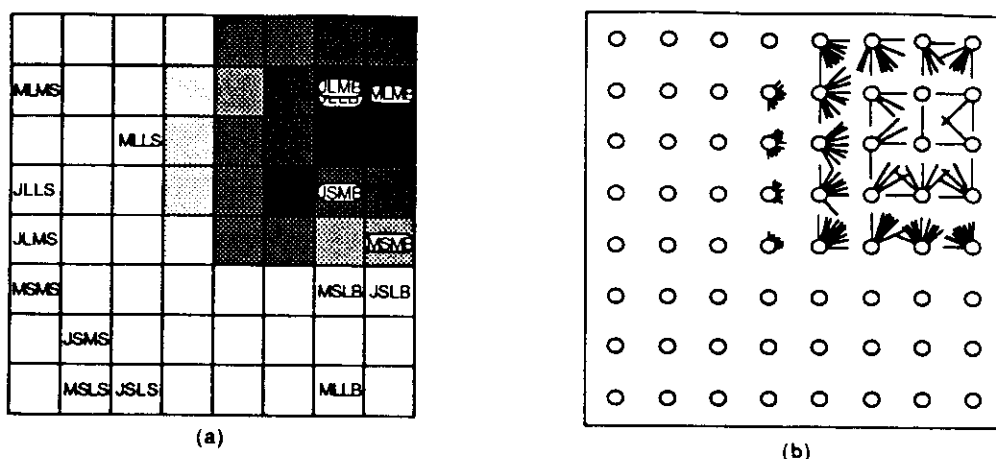


Figure 8.5: **Creating a trace for a fancy-restaurant-JLMB.** The trace is created within a radius 2 neighborhood of the maximally responding unit, labeled JLMB (see figure 7.6a for role-binding abbreviations). The response pattern is shown in (a), and the excitatory lateral weights are plotted in (b).

words, the unit closest to the input vector is found, and the output activity is concentrated and scaled between 0 and 1 within a neighborhood of this unit. This is an efficient abstraction of the process of focusing the activity with lateral inhibition. The response is a well-localized and focused activity bubble around the initial activity maximum, with some of the variation within the bubble still retained. Settling iterations are avoided, and the mechanism does not require additional parameter optimization like lateral inhibition. This is a general approximation technique, and it will be used also in the implementation of the lexicon (chapter 9).

Figure 8.5 shows the response patterns and the traces created for two fancy-restaurant story representations. These figures also illustrate two additional modifications to the ideal trace map mechanism. On role-binding maps, square neighborhoods are used instead of round ones because the self-organization of the maps was based on square neighborhoods. This is mostly for consistency since the shape of the neighborhood is not crucial. As often is the case with feature maps, the boundary needs special attention. The neighborhoods are shifted away from the boundary, so that each trace is coded by an equal number of units. This makes each trace initially equally strong in the memory, no matter where it is located.

8.2.2 Retrieval

The retrieval process also needs to be modified to make it robust on more structured maps. In the ideal process, if the localized response to the cue overlaps with the trace, the positive connections within the trace pull the activity to the center of the trace. Now the initial response to the cue is less localized, and similar vectors are sometimes mapped far apart. Retrieval cannot be based on simple overlap. Instead, the initial response must cover the entire map, and the retrieval depends more on properly combining the external input activity and the lateral activity in the settling process.

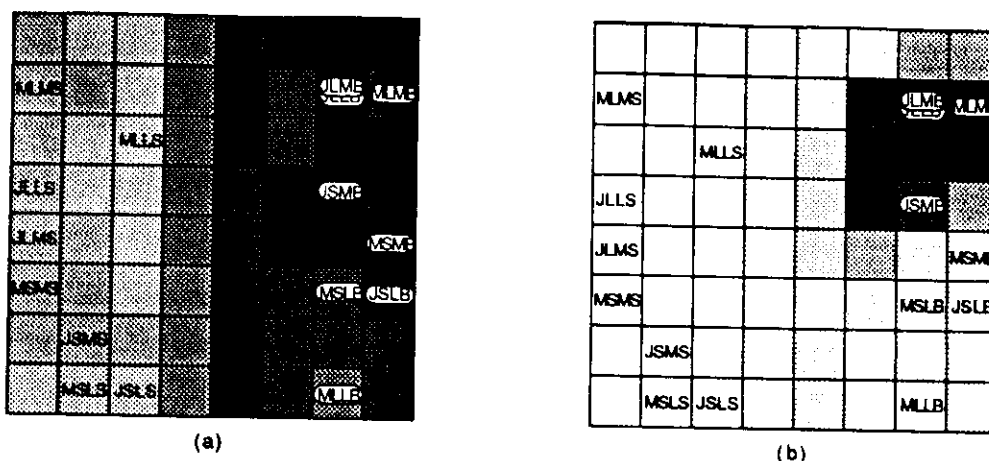


Figure 8.6: Simple retrieval of fancy-restaurant-JLMB. The representation of JLMB is retrieved with a cue initially mapped on the unit labeled JSMB. Initial response pattern is shown in (a), and the stable activity pattern after 50 settling iterations in (b).

A retrieval example is shown in figure 8.6. When the cue vector is close to the stored vector, the external input activity will be high at the center of the trace. This activity is combined with the strong positive lateral connections within the trace, increasing the activity within the trace and turning it off elsewhere. The final response settles around the center of the trace.

If there are multiple traces in the same highly active area (figure 8.7), they compete for the cue (figure 8.8). In addition to increasing the activity at the center of the trace, each unit within one trace inhibits the other trace. Usually the later trace covers more units, and is eventually able to turn the other traces off.

However, if the cue vector is dissimilar to the stored vector, the trace is initially covered only with low activity (figure 8.9). The positive lateral connections are overcome by inhibition from the highly active areas, and the activity on the whole map is turned off during the next time cycle. As in the ideal case, the pattern oscillates between initial response and zero activity, indicating that nothing can be retrieved.

8.3 Storage and retrieval from episodic memory

The mechanisms for memory organization and storage and retrieval of traces have been discussed separately above. This section describes how the hierarchical organization and the trace feature maps work together as episodic memory.

8.3.1 Storing representations

A representation to be stored in the episodic memory is presented to the hierarchical feature map system. The hierarchy classifies it as an instance of a particular script, track and role binding. In other words, the hierarchy determines the appropriate role binding map

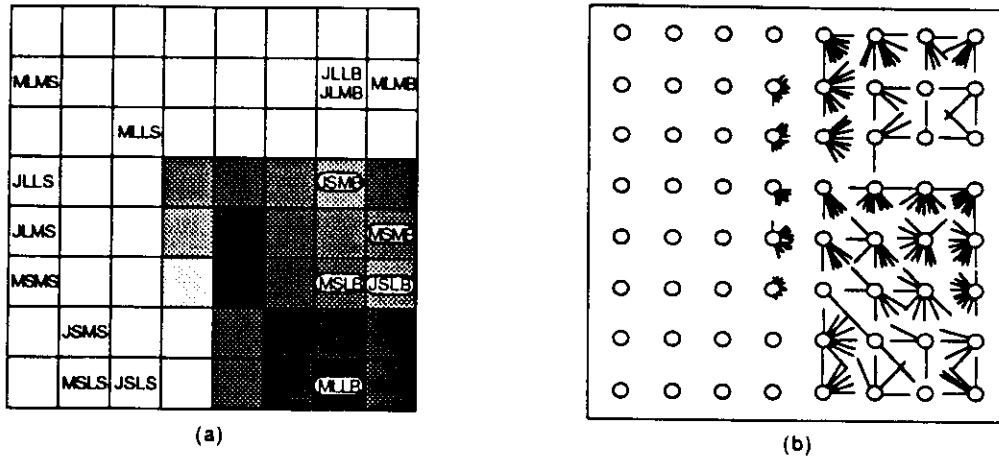


Figure 8.7: Adding a trace for fancy-restaurant-MLLB. The new trace (neighborhood radius 2) is created around the unit MLLB, stealing some units from the earlier trace.

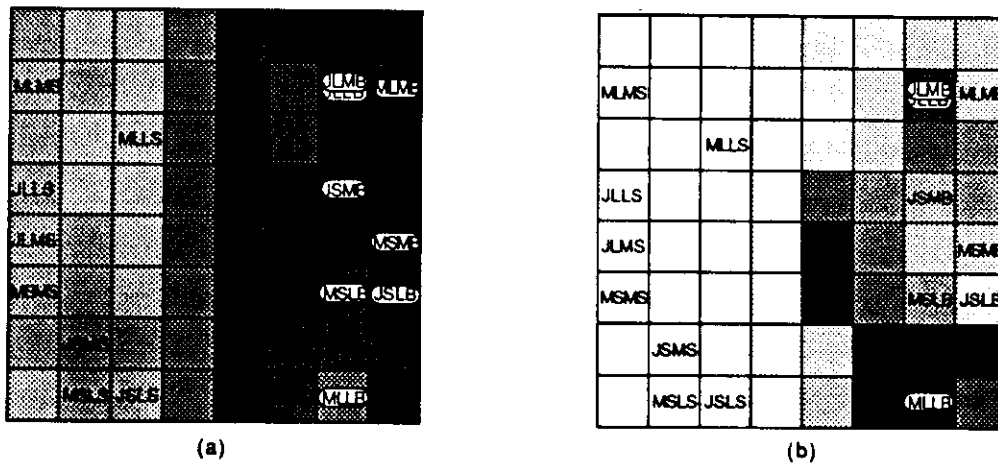


Figure 8.8: Retrieval with competing traces. The same cue, JSMB is presented. The later trace MLLB is stronger on the map, and even though it is further away from the cue, it is retrieved instead of JLMB. Initial response is shown in (a), stable activity pattern after 50 settling iterations in (b).

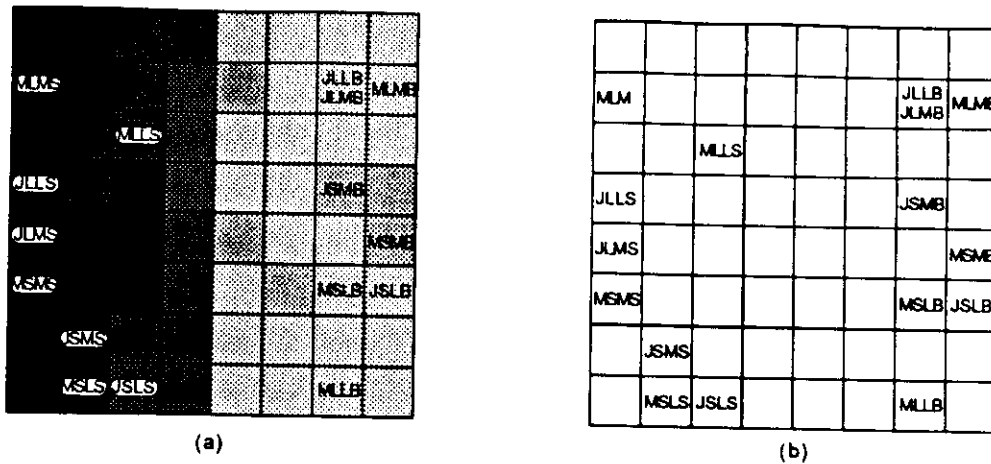


Figure 8.9: **Unsuccessful retrieval.** The cue vector is mapped on the unit MSMS, which is on the left, i.e. the **small tip side** of the map (the last letter in the label indicates the size of tipe, S=small, B=Big) . Both traces JLMB and MLLB are located on the other side. In this case, neither one of them is retrieved, and the activity oscillates between the initial pattern (a) and zero response (b).

and the location on that map. The trace feature map mechanism then creates a memory trace at that location using lateral connections between units.

For example, the representation vector of RFJLMB (i.e. a fancy-restaurant story with the role bindings John, lobster, MaMaison, big) is recognized as an instance of the restaurant script and the fancy-restaurant track (figure 8.10). At the same time, the vector is compressed into a few most significant components (figure 7.3). The role-binding map for fancy-restaurant develops a localized response around the unit JLMB (figure 8.5a), and the trace is coded in the lateral weights within the response (figure 8.5b).

8.3.2 Basic retrieval

A story is retrieved from the memory by giving it a partial story representation as a cue. Unless the cue is highly deficient, the map hierarchy is able to recognize it as an instance of the correct script and the track, and form a reasonable guess about the role binding. The trace map mechanism then determines whether an appropriate trace exists and settles to the accurate role binding. The stored vector is obtained from the weights of the image units at the three levels.

For example in figure 8.11, a representation of RFJSMB (fancy-restaurant with John, steak, MaMaison, big) is used as a cue. The cue is a fairly good approximation of the stored vector, except the food is incorrect. The hierarchy classifies this vector as a representation for fancy-restaurant story (compressing it along the way), and the initial response on the role-binding map is centered around JSMB (figure 8.6a). This is close enough to the trace of JLMB, and the activity is drawn to the center of the trace (figure 8.6b). The complete story representation is retrieved from the weight vectors of the maximally responding units at the script, track, and role binding levels, as illustrated in figure 7.9.

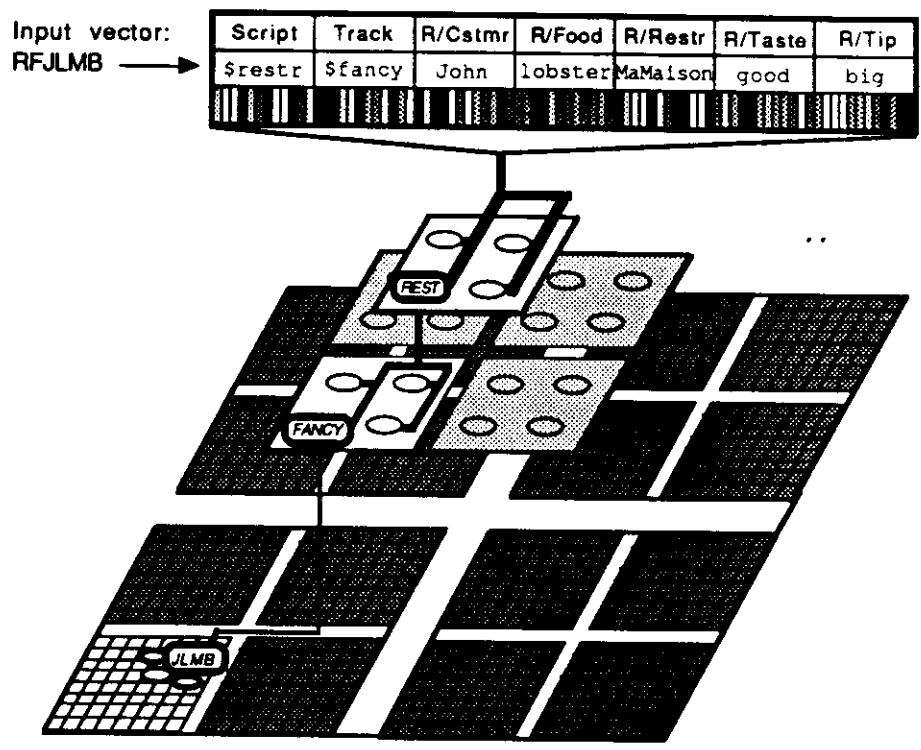


Figure 8.10: **Storing RFJLMB.** The input representation is recognized as an instance of the restaurant script and the fancy-restaurant track, and the trace is created around the role-binding unit JLMB (only a few units are shown at the bottom level map).

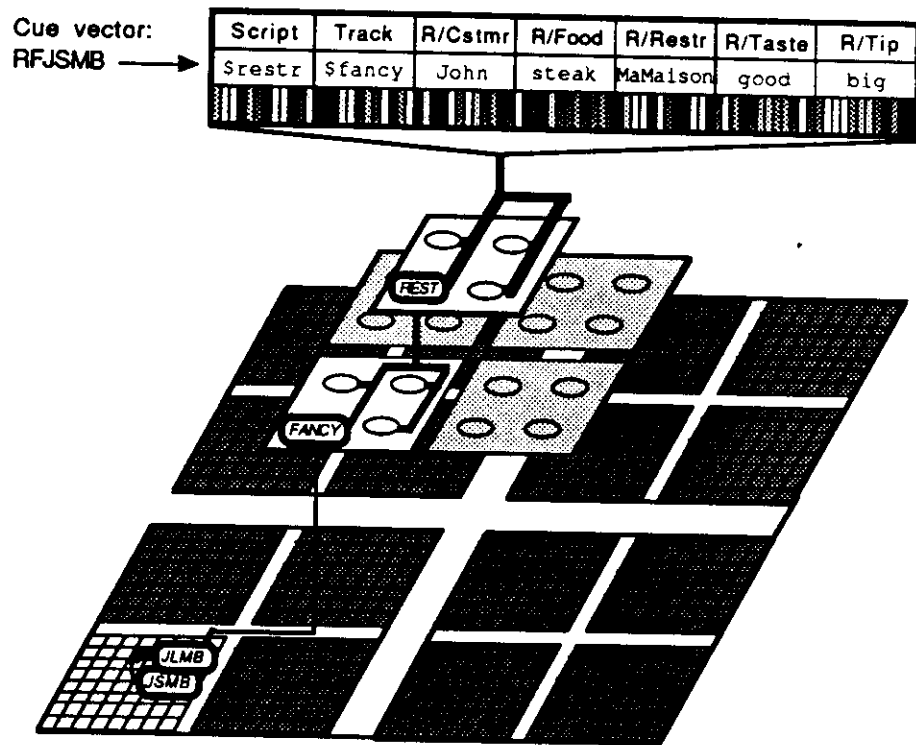


Figure 8.11: Retrieving RFJLMB with RFJSMB as a cue. The cue representation is classified into the role-binding map of fancy-restaurant, where the initial activity is centered around the unit JSMB. The trace feature map mechanism moves the activity to JLMB, the center of the trace.

Mode	All words	Instances	$E_i < 0.15$	E_{avg}
Storage	99	99	99	.0055
Retrieval	99	99	99	.0055

Table 8.1: **Feasibility of retrieval.** Each one of the 96 test story representations was stored in the episodic memory by itself, and retrieved with itself as a cue. The accuracies are the same, indicating that whatever was stored was also retrieved.

The taxonomy was formed by presenting the hierarchy with only complete script representations. The system has the nice property that incomplete representations are automatically completed as a side effect of their classification. In the same manner, incorrect information in the cue is automatically corrected in the retrieval. The cue is a soft constraint for retrieval, and the memory finds the trace which best matches the cue.

The memory can also recognize when the cue is too far from anything in the memory. For example, RFMSMS (fancy-restaurant with **Mary, steak, MaMaison, small**) has different customer, food, and tip amount than the stored trace. This story is classified to the fancy-restaurant map, but the initial activity on that map is too far from the trace and does not settle (figure 8.9).

As discussed in section 8.1.4, the different memory effects make it hard to analyze the performance of the feature map mechanism, and the same concerns the performance of the episodic memory as well. A minimum requirement is that under optimal conditions it should be possible to retrieve any stored vector. The episodic memory satisfies this requirement, as can be seen from table 8.1. The accuracy of retrieval is exactly the same as the accuracy of storage, indicating that every vector was retrieved exactly as it was stored.

8.3.3 Ambiguous retrieval

When the cue is so deficient that it is not possible to determine the correct script and track, the process becomes slightly more complicated. For example, the question **What did John eat?** specifies that **John** was the main actor in a restaurant story, but any track could be possible.

In cases like the above, it is necessary to search for the appropriate trace on all possible role-binding maps. A search threshold parameter is used to decide which maps are feasible. With the above question, the restaurant unit on the top-level map responds with 0.4 (distance from the cue), while the other units are all further than 2.8. With the top-level search threshold at 1.0, only the restaurant maps will be considered. This decision is fairly robust.

However, the different track units vary between 1.3 and 2.2, none of them significantly closer to the cue than the others. With the search threshold at 2.5, all maps will be looked at in parallel. Setting the search threshold for the middle level is trickier, but fortunately the retrieval process robustly can find the correct trace fairly robustly.

If the question determines a unique story in memory, only one of these submaps leads to

a trace at the bottom level. The activity on the other branches does not settle, indicating "not found". If several traces are found, the one closest to the cue will develop the highest activity, and will be selected. For example, if there is a fancy-restaurant trace with John as the customer, it responds stronger to the above question than e.g. a fast-food-restaurant trace with Mary.

The process works the same in truly ambiguous cases, where there are multiple correct matches in the memory. For example, if there is a fancy-restaurant trace and a fast-food-restaurant trace, both with John as the customer, DISCERN selects the one with the strongest response and returns the weights of this trace as the successfully retrieved vector. Several examples of ambiguous retrieval are given in section 10.2.

The above mechanism is simple and works reasonably well. Currently it takes place as part of the simulation control (code in appendix D.1.5), but it might be possible to do it with neural mechanisms. If the neighborhood selection is implemented with lateral inhibition, the thresholding would take place automatically. Selection of the strongest response could be implemented with winner-take-all.

However, more elaborate control of retrieval and certain interesting high-level memory phenomena require a higher-level control process. For example, the memory currently does indicate (by oscillation) when there is no appropriate trace in the memory. The system should generate an appropriate reply sentence, e.g. **Cannot answer the question** in such situations. In ambiguous retrieval it would be natural to ask for additional information: **The question is ambiguous**. When several traces remain active in the settled activity pattern, a combination of them could be retrieved instead of the maximally responding one.

These decisions can only be made by a higher level process, which monitors the state of the memory and the retrieval. It would be simple to implement them with symbolic control, but we have resisted the temptation, and left it as a future problem to be worked out with neural networks (see section 13.6).

Chapter 9

Lexicon

Processing in DISCERN is based on semantic concept representations, which stand for the meanings of words. The components of DISCERN discussed up until now only deal with concepts. However, DISCERN communicates with the outside world using orthographic symbols. The lexicon has the task of mapping the symbols to their meanings. It forms the interface between the internal world of DISCERN and the environment. The lexicon also has to be able to model ambiguities in this mapping, and provide a mechanism for disambiguation.

The lexicon architecture is based on feature maps. The symbol and concept representations are laid out on separate maps and translation between them is implemented through associative connections. The connections learn a many-to-many mapping from one distributed representation space to another. In the maps, several representations can be active at the same time, making it possible to represent ambiguity. Priming of alternatives can be implemented by combining associative activation with ordinary map activation.

9.1 Overview of the architecture

The lexicon consists of two parts: the lexical memory, which contains the representations for the orthographic symbols, and the semantic memory, which consists of representations of concepts. Both the symbols and concepts are represented distributively, i.e. as vectors of gray-scale values between 0 and 1.

The lexical and semantic memories are implemented as feature maps (figure 9.1). As discussed in chapter 6.1, the maps lay out each high-dimensional representation space on a 2-D area so that the similarities between words become evident. Words whose orthographic forms are similar (e.g. **BALL**, **DOLL**) are represented by nearby units in the lexical map. In the semantic map, words with similar content (e.g. **livebat**, **prey**) are mapped near each other.

The lexical map is densely connected to the semantic map with one-way associative connections (figure 9.1). The connections exist in both directions (the connections from semantic to lexical map are omitted from the figure). A localized activity pattern representing a symbol in the lexical map will cause a localized activity pattern to form in the semantic map, representing the meaning of the symbol (figure 9.1). Similarly, an active meaning activates a symbol in the lexical map. The lexicon thus transforms an orthographic input representation into a semantic output representation, and vice versa, and serves as an input/output filter for language processing.

The lexical and semantic maps are organized and the associative connections between

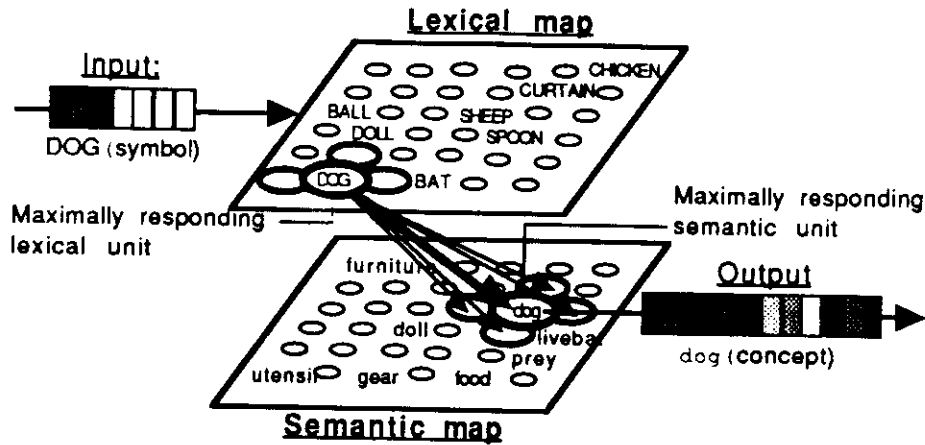


Figure 9.1: Lexical and semantic feature maps. The lexical input word DOG is transformed into the semantic representation of dog. The representations are vectors of real values between 0 and 1, shown by gray-scale coding. The size of the unit indicates the strength of its response. Only a few strongest associative connections are shown.

them are formed simultaneously in an unsupervised learning process, by presenting the system with co-occurring symbols and concepts.

9.2 Representation of lexical symbols

The architecture does not pose many restrictions to the lexical representations. They could be based on any modality. Since AI systems typically communicate with their environment using text, the orthographic modality was chosen for DISCERN. The central requirement for the orthographic representations is that they should reflect the visual similarities of the written words.

A simple encoding scheme was devised to build such distributed representations. Each character was given a value between 0 and 1 according to its darkness, i.e. how many pixels are black in its bitmap representation in the Macintosh Geneva font. The darkness values of the word's characters were then concatenated into one representation vector. The encoding code is listed in appendices C.3.2 and E.4.1, and examples of data are shown in C.3.3 and E.4.2. The encoding scheme is extremely simple and leaves out most of the visual qualities. But it does ground the representation into the sensory experience, and it turns out to be completely adequate for our purposes. Each written word symbol has a unique representation, and similar symbols have similar representations.

9.3 Properties of the lexicon model

The story data for DISCERN is not ideal for demonstrating the various properties of the lexicon model. The vocabulary in the stories is quite large and consists of very specific words. There are no ambiguities nor interesting similarities in the use of the words. In this section we

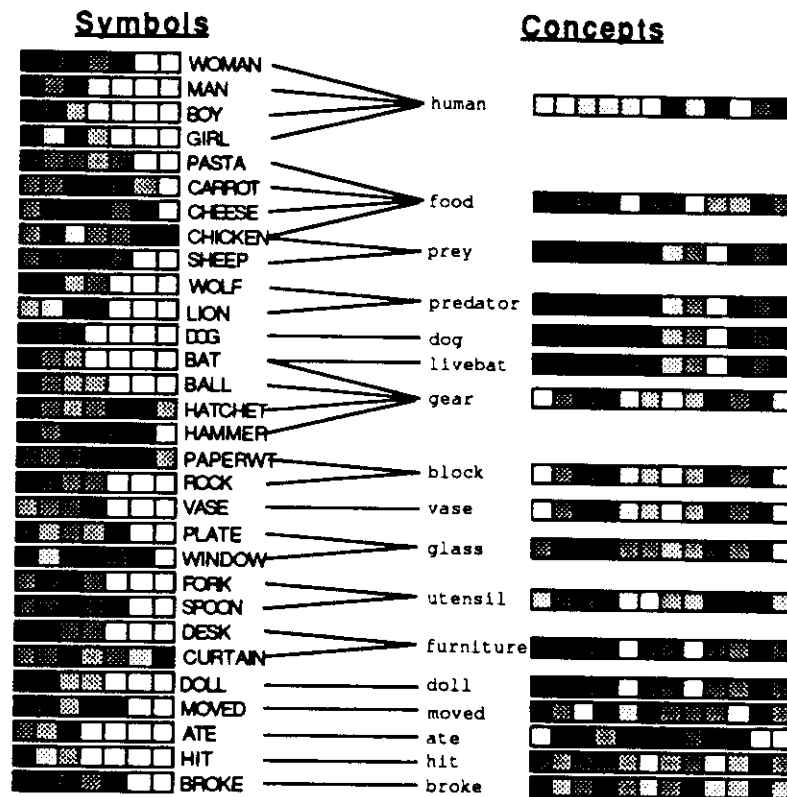


Figure 9.2: **The training data.** The symbol representations code the orthographic forms of the words, while the concept representations stand for distinct meanings. Gray-scale boxes indicate component values within 0 and 1. The connections depict the mapping between the symbols and their meanings. Many concepts map to several synonymous lexical symbols, and the homonymous symbols CHICKEN and BAT map to two distinct concepts each.

will present a version of the lexicon which was developed for sentence processing data. This data contains several interesting ambiguities and more comprehensive word categorization, and better brings out the properties of the model. The actual configuration of the DISCERN lexicon, which has the same architecture but different training, is presented in section 9.4. Both the sentence processing lexicon and the DISCERN lexicon were developed with the same training code, listed in appendix C.3.1.

9.3.1 Sentence training data

The training data was obtained from the case-role assignment example of chapter 4. Samples of data and the code for generating it are listed in appendix E.4. The full vocabulary of the original data (table 4.2) was used for the symbols. The visual form of each word was encoded into its distributed representation (figure 9.2).

The concept representations were obtained by training the nonrecurrent case-role assignment network (figure 4.4) with the unique concepts in the data. In other words, only one

Category	Semantic words
animal	prey predator livebat dog
fragileobj	glass vase
breaker	gear block
hitter	gear block vase
possession	gear vase doll dog
object	gear block vase glass food furniture doll utensil
thing	human animal object
verb	hit ate broke moved

Table 9.1: **Concept categories.** The same sentence templates with the same slots and filler categories are used to generate the sentence data as before (table 4.2), but now the categories consist of semantically distinct concepts.

representation was developed for each equivalence class (section 4.5.3), and the occurrences of the ambiguous words *chicken* and *bat* were replaced by the unambiguous concepts *food*, *animal* and *gear* (table 9.1).

The concept representations that result reflect the use of the semantic words (figure 9.2). Words belonging to the same category have a number of uses in common, and their representations are similar. The total usage is different for each word, and consequently, their representations are different. They stand for unique meanings.

The lexical and semantic word representations were formed prior to organizing the lexicon. During the FGREP process, the external training supervisor maintained an abstraction of the semantic component of the lexicon; the semantic representations were not stored on a feature map. The lexicon was trained with the final representations only, and they remained fixed throughout the lexicon simulations. This is the most efficient way to develop the representations and to organize the lexicon (possibilities for combining these processes are discussed in section 13.3.2).

The lexicon was organized in 150 epochs, i.e. by presenting each lexical/semantic representation pair (figure 9.2) to the appropriate map 150 times in random order. The lexical and semantic maps are organized independently, albeit simultaneously, so that associative connections between them can be developed at the same time (see next section). The same adaptation gain $\alpha(t)$ was used for both maps and the associative connections (equations 6.6 and 9.2). The $\alpha(t)$ was linearly decreased from 0.1 to 0.05 during the first 50 epochs, then to 0.0 during the remaining 100 epochs. At the same time, the neighborhood radii on the two maps were decreased from 4 to 1 and then from 1 to 0.

9.3.2 Lexical and semantic maps

In the self-organizing process, the lexical and semantic representations become stored in the weights of the units. For each e.g. lexical symbol, there is an image unit in the lexical map, and this unit's weight vector equals the lexical representation of that word. The weight vectors of the intermediate units represent combinations of representations. For example, an unlabeled semantic unit between *dog* and *predator* would have features of both domestic and carnivorous animals.

Both maps exhibit hierarchical knowledge organization (figures 9.3 and 9.4). Large areas

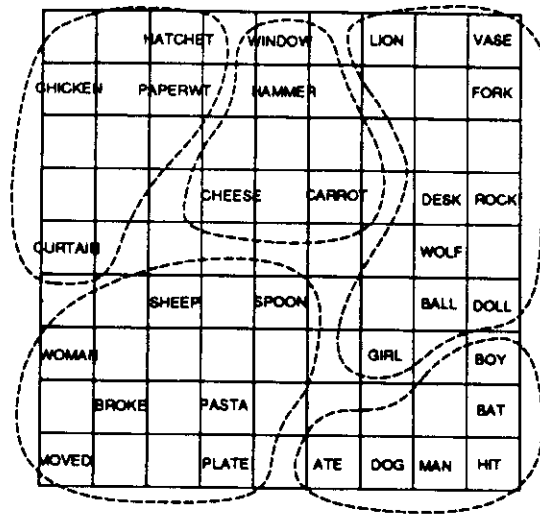


Figure 9.3: **The lexical map.** Each unit in the 9×9 network is represented by a box in the figure. The labels indicate the image unit for each symbol representation. The map is divided into major subareas according to word length.

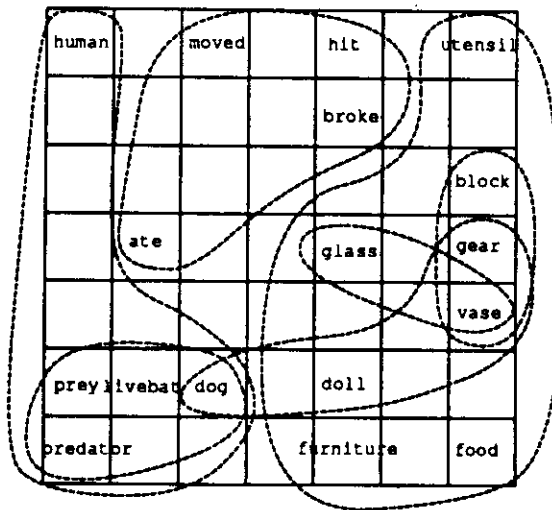


Figure 9.4: **The semantic map.** The labels on this 7×7 map indicate the maximally responding unit for each concept representation. The map is organized according to the semantic categories (table 4.2).

are allocated to different categories of words, and each area is divided into subareas with finer distinctions. The lexical map becomes mainly organized according to the word length. There are separate, adjacent areas for symbols with 3, 4, 5, 6 and 7 characters. Within these areas, similar words are mapped near each other. For example, BAT is mapped between BOY and HIT, DOLL is mapped next to BALL etc.

The semantic map has three main areas: verbs, animate objects and inanimate objects. Finer distinctions reveal the semantic categories of table 4.2. For example, there are subareas for hitters, possessions and fragile-objects, with vase, which belongs to all these categories, in the center. As discussed in chapter 4, the categorization was not directly accessible to the system at any point. It was only manifest in the sentences that were input to the FGREP-mechanism. The categories were extracted by FGREP, coded into the representations, and finally made visible in the semantic feature map. The final map reflects both the syntactic and semantic properties of the words.

As discussed in section 6.1, feature maps represent the most frequent areas of the input space in greater detail. This can be clearly seen in the word maps. For example, the semantic representations for the different animals are very similar (figure 9.2), yet they accommodate a large area on the semantic map. Also, the self-organizing process automatically develops the most descriptive dimensions for the map. As a result, the areas on the map are likely to have complicated and intertwined, rather than linear shapes.

9.3.3 Word associations

The lexical symbols do not correspond one-to-one to concepts. Some words have multiple meanings (homonyms), and sometimes the same meaning can be expressed with several different symbols (synonyms). The mapping between the lexical and semantic representations is many-to-many.

The training data contained several such ambiguities (figure 9.2). The lexical symbol CHICKEN could mean a living chicken or food. Similarly, BAT could be a baseball bat or a living bat. There were also several groups of synonymous words in the data. MAN, WOMAN, BOY, GIRL all have the same meaning human, predator could be WOLF or LION etc. The many-to-many mapping between the symbols and their meanings is implemented with associative connections between the lexical and semantic maps.

The lexical map is fully connected to the semantic map with unidirectional associative connections (figure 9.1). There is a connection from each unit in the lexical map to each unit in the semantic map, and from each unit in the semantic map to each unit in the lexical map. The connection weight indicates the strength of the association. The weights are stored as associative output weight vectors per each unit.

The lexical and semantic feature maps and the associative connections between them are organized at the same time. The orthographic pattern for the word is presented to the lexical map, which develops a localized activity pattern around the image unit. In our simulation,

the output η_{ij} of unit (i, j) is

$$\eta_{ij} = \begin{cases} 1 - \frac{\|x - m_{ij}\| - d_{min}}{d_{max} - d_{min}} & \text{if } (i, j) \in N_c \\ 0 & \text{otherwise} \end{cases} \quad (9.1)$$

where m_{ij} is the unit's weight vector, x is the input vector, N_c is the neighborhood around the maximally responding unit, d_{min} is the smallest and d_{max} is the largest distance of x to a weight vector in the neighborhood (see functions `compute_responses` and `find_max_assoc` in appendix C.3.1). This technique approximates the focused pattern of activity produced by lateral inhibition (as discussed in section 8.2.1), and forms a nice concentrated activity pattern around the maximally responding unit. Ordinary feature map adaptation (i.e. self-organization) takes place within the neighborhood.

At the same time, the semantic pattern for the same word is input to the semantic map, which develops a localized response, and the feature map weight vectors in this map are adapted within the response. At this point, both maps display localized patterns of activity. The lexicon learns to associate the lexical symbol with its meaning through Hebbian learning. The weights between active units are increased proportional to their activity:

$$\Delta l_{ij, kh} = \Delta s_{kh, ij} = \alpha(t) \eta_{L, ij} \eta_{S, kh} \quad (9.2)$$

where $l_{ij, kh}$ is the weight from the lexical unit (i, j) to the semantic unit (k, h) , $s_{kh, ij}$ is the weight from the semantic unit (k, h) to the lexical unit (i, j) , and $\eta_{L, ij}$ and $\eta_{S, kh}$ indicate the activities of these units (see function `modify_assoc_weights` in appendix C.3.1). The associative weight vectors are then normalized, which in effect decreases the weights on all nonactive output connections of the same unit. This corresponds to redistribution of synaptic resources, where the synaptic efficacy is proportional to the square root of the resource [Miikkulainen, 1987]. Initially the activity patterns are large, and associative weights are changed in large areas. As the two maps become ordered, the associations become more focused.

The lexicon was trained by simultaneously presenting pairs of lexical symbols and their semantic counterparts from figure 9.2. The final associative connections form a continuous many-to-many mapping between the two maps. Unambiguous words have focused connections (figures 9.5a and 9.6b). If a symbol has several meanings, or one meaning can be expressed with several synonyms, there are several groups of strong connections (figures 9.5b and 9.6a). Units located between image units tend to combine the connectivity patterns of nearby words (figure 9.6a).

9.3.4 Transforming representations

A symbol is transformed to its semantic counterpart (and vice versa) through the associative connections. For example in figure 9.1, the lexical representation of DOG is input to the lexical map, which forms a concentrated activity pattern around the unit labeled DOG. The activity propagates through the associative connections (figure 9.5a) to the semantic map, where a localized activity pattern forms around the unit labeled dog. The semantic

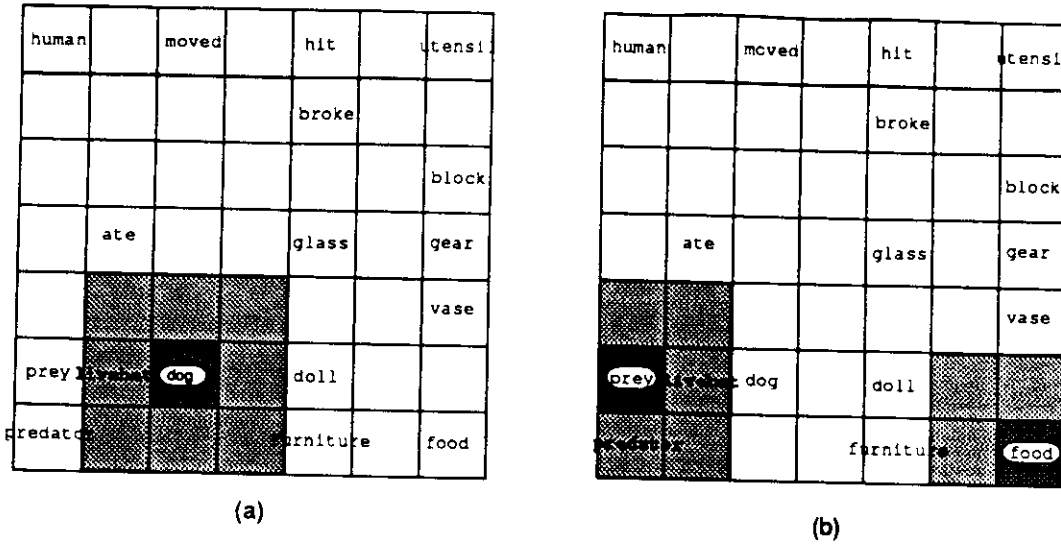


Figure 9.5: **Sample lexical → semantic associative connections.** The darkness of the box indicates the strength of the connection from the lexical unit to the semantic unit. Figure (a) depicts connections from the lexical unit **DOG**, and (b) from the lexical unit **CHICKEN**. The strongest connections concentrate around the semantic image units. **CHICKEN** has two possible interpretations, **food** and **prey**.

representation for dog is now output through the weight vector of this unit. In a similar fashion, a semantic representation can be transformed to its lexical counterpart. The associative connections are different in the two directions, but the same feature map weight vectors are used for both input and output.

The behaviour of the system is very robust. Even if the input pattern is noisy or incomplete, it is usually mapped onto the correct unit. Even if this does not happen, the associative connections of the intermediate units provide a mapping that is close enough, so that the correct meaning or symbol can be retrieved with *top-down priming*.

9.3.5 Priming

When an ambiguous lexical or semantic representation is input to the lexicon, all possible meanings (or symbols) are activated at the same time (figures 9.5b and 9.6a). A top-down priming mechanism is employed to select the correct representation. In addition to the associative activity, the map receives priming activation through its input connections. The activities add up, selecting one of the possible interpretations. If the priming arrives after a short delay, all alternatives are briefly active before one of them is selected. This complies with experimental results [Swinney, 1979], which indicate that all meanings of ambiguous words are activated upon reading the word.

Expectations generated by the sequential FGREP mechanism provide a possible source for priming activation. After reading *The wolf ate the*, the FGREP network generates a strong expectation for *prey* (section 4.6.4). When the lexical symbol **CHICKEN** is read in, both the **food** and **prey** units are initially equally active in the semantic map (figure 9.5b). The

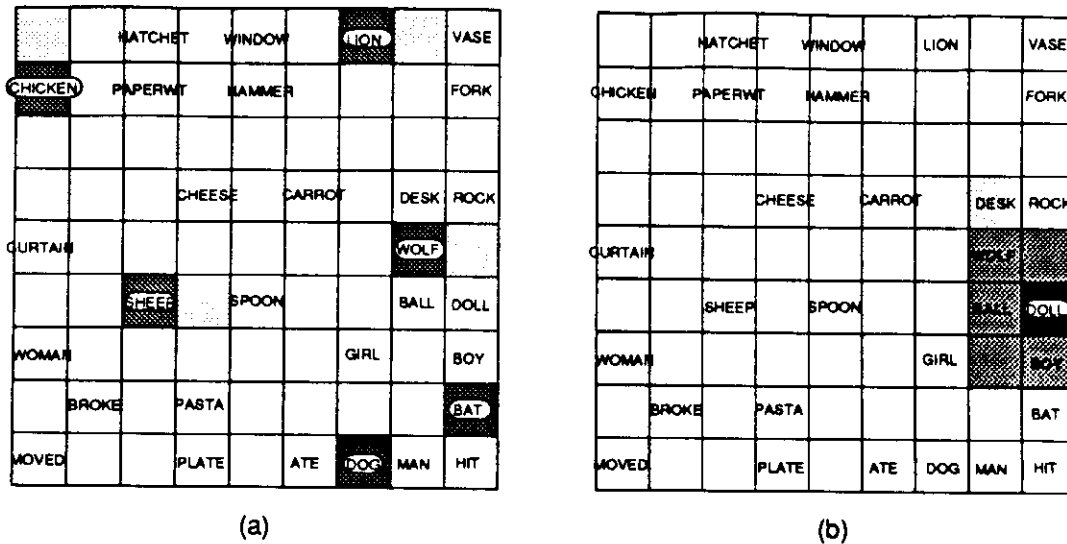


Figure 9.6: Sample semantic \rightarrow lexical associative connections. In (a), the connections from the intermediate unit between dog, livebat, predator and prey are shown. Possible output symbols include all animal names CHICKEN, SHEEP, WOLF, LION, BAT and DOG. In (b), weak connections from doll to nearby units might cause BALL to be output instead of DOLL in noisy conditions.

expectation pattern, which is close to the representation for prey, is input to the semantic map and summed up with the activity propagated through the associative connections. As a result, the prey unit receives the strongest activity and becomes selected.

The weights on the associative connections learn to represent statistical likelihoods of the associations. A very frequently active connection becomes much stronger than a rare connection. For example, if most of the occurrences of CHICKEN in training the lexicon would have been paired up with prey, the CHICKEN unit would tend to activate the prey unit much more than the food unit. By default, the prey meaning would be selected, and stronger priming for food would be required to override it.

9.3.6 Errors

The lexicon architecture is well suited for modeling dyslexic performance errors. If the system performance is degraded e.g. by adding noise to the connections, two types of input errors and two types of production errors are observed.

In input, a lexical representation may be mapped incorrectly onto a nearby unit in the lexical map. This corresponds to reading or hearing the word incorrectly. For example, DOLL may be input as BALL (figure 9.3). The activity in the lexical map may also propagate incorrectly to a nearby unit in the semantic map, in which case e.g. CHICKEN would be understood semantically as livebat (figure 9.5b).

Analogously in production, a semantic input representation can be classified incorrectly, and a word with a similar but incorrect meaning is produced. For example, if the semantic pattern for block is accidentally mapped on vase (figure 9.4), the output reads VASE instead

of, say, PAPERWT. Or, the activity in semantic map may be propagated incorrectly to the lexical map, and a word with a similar surface form but different meaning is output. This means generating BALL instead of DOLL (figure 9.6b).

Errors of this kind occur in noisy, stressful or overload situations in normal human performance. They are also documented in patients with deep dyslexia [Coltheart et al., 1980; Caramazza, 1988]. The observed visual and semantic paralexical errors can be explained by above mechanisms, lending support to the lexical/semantic feature map architecture.

With priming, it is also possible to model another type of performance error: the Freudian slip. This occurs when very strong semantic priming interferes with the output function. For example, if doll is input to the semantic map, together with simultaneous priming for gear, the activity is propagated through the associative connections of both. As a result, the symbol BALL receives the strongest activation, and is output instead of DOLL. The output symbols are similar, but the meaning of BALL reveals the semantic priming.

9.4 The lexicon in DISCERN

9.4.1 Training the lexicon

The architecture and mechanisms described in the previous sections were applied to the vocabulary of script-based stories to build a lexicon for DISCERN. As mentioned before, this vocabulary is larger but consists of words with very specific use, without ambiguities and with less obvious semantic categories. An interesting addition is the inclusion of the IDs. The test vocabulary contains two instances of each word prototype, which brings up interesting issues in training the lexicon.

It is quite plausible to organize the semantic map with randomly set IDs¹. In this case the semantic map would form a mapping of the space of all possible instances, just like the role-binding maps lay out the space of all possible role bindings. Within the resolution limits, every word instance that actually occurs in the performance phase would have a unique location in the semantic map.

In principle, we could also train the symbol map this way, by presenting it with random examples of character strings, or more plausibly, with random examples of strings that are possible in the particular language. The map network would form a mapping of the symbol space of the language. The third phase of the training would then be to present the maps with the actual symbols and concepts (with fixed IDs) that occur in the language, and self-organize the associative connections between them. In training the lexicon this way, the space of possible concepts and possible symbols in the language is first organized separately, and then a mapping is formed between the symbols and concepts that actually occur in the language.

While conceptually this training scheme makes sense, it is hard to implement in practice. The maps, especially the symbol map, need to be extremely large to represent all possible

¹Again, it is more efficient to train the lexicon only with the final semantic representations instead of organizing and re-organizing the semantic map along while these representations are being developed.

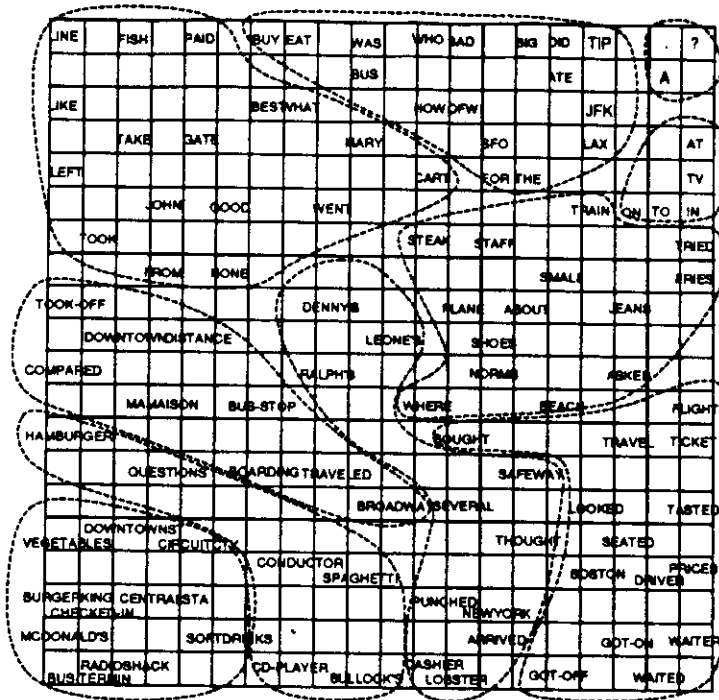


Figure 9.7: The lexical map of DISCERN. The map becomes organized according to word length.

symbols even with modest accuracy, and unfortunately, the symbol representation is very sensitive to errors. Even minor inaccuracies in the representation may change the symbol's meaning completely, as in e.g. WAITER → WAITED.

The alternative is to not insist on representing the whole space, but organize the maps with the actual symbol-concept pairs only. Even in this case the map tends to represent the areas *between* the training examples continuously, i.e. it develops a partial representation of the space. Most importantly, the representations for the training examples are now accurate.

The DISCERN lexicon was trained with the actual symbol-concept pairs. Examples of the training data and the simulation specification file are listed in appendix C.3.3, and parts of the program used to generate the data are listed in appendix C.3.3. The same ID values were used as in testing the episodic memory, i.e. $(0.75, 0.25$ and $0.25, 0.75)^2$. The maps consisted of 20×20 units. The maps and the associative connections between them were organized simultaneously in 500 epochs (figures 9.7 and 9.8). The maps display similar organizing principles as before. The symbol map divides into main regions according to the word length. The different instances of the same word prototype are mapped next to each other in the concept map.

²It turns out that more extreme values would place the instances too far apart, when Euclidian distance is used as the similarity metric. The distance of two instances with ID values (1,0 and 0,1) is equal to an average component difference 0.41, which is far above the average difference, and the instances would be less similar to each other than to a random vector in the representation space. This problem could be overcome with a different similarity metric, but using less extreme values is a simpler solution, and also consistent with the properties of the episodic memory.

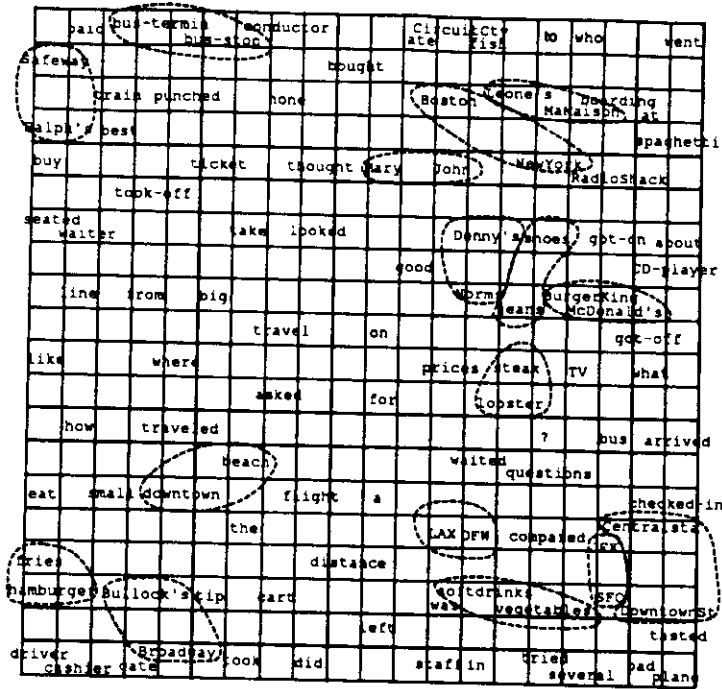


Figure 9.8: The semantic map of DISCERN. The different instances of the same word prototype are mapped next to each other.

Initially, each pair was presented once per epoch, which resulted in a map where some words were mapped on the same units. This is a common problem in forming feature maps of discrete vector spaces. The maps are small, and the self-organizing process is not always accurate enough to separate the items within a small number of units. Rather than increasing the map size, additional copies of hard-to-separate items were included in the training set. This forced the process to allocate more units to that part of the input space, and the resulting map nicely separates all words.

In terms of performance it does not matter what distribution the map is trained with as long as the final map provides a unique representation for each word. The training distribution has an effect on the organization of the mapping, allocating larger areas for more frequent parts of the word space. The most plausible solution would be to train the maps with the distribution in the task. However, some words occur so rarely that the separation becomes even harder than in the uniform case. Adding additional emphasis on the hard-to-separate or seldom occurring meanings and symbols is a plausible technique, which allows us to use smaller maps and less expensive training.

The final accuracy of the lexical and semantic map representations and the output representations of the lexical \rightarrow semantic and semantic \rightarrow lexical mapping are presented in table 9.2 (the testing routines are part of the training program, see appendix C.3.1; the same data as the lexicon was trained on was also used to test it). Once the map has settled and uniquely represents each word, the representations can be made arbitrarily accurate by continuing training with zero neighborhood size, i.e. adapting only the image unit and its associative connections. This is how the almost perfect accuracy of table 9.2 was achieved.

Component	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical	100	100	100	.0000
Semantic	100	100	100	.0002
Assoc L→S	100	100	100	.0002
Assoc L←S	100	100	100	.0000

Table 9.2: **Accuracy of the DISCERN lexicon.** There were two clones for each word prototype, with ID values 0.75,0.25 and 0.25,0.75. The numbers indicate accuracy of the word representations on the lexical and semantic maps, and the correctness of the associative connections after 500 epochs. The mapping can be made arbitrarily accurate by training with zero neighborhoods.

9.4.2 Filtering out noise

If the input representations to the lexicon are fairly accurate, it is easiest to use zero neighborhoods also during the performance phase. An input representation is presented to the lexicon, the maximally responding unit is found, and the activity is propagated through the associative connections of this unit. The unit with the highest activity in the other map after propagation is taken as the output image, and its weight vector constitutes the output representation. The process is already quite robust against noise, because a noisy input vector is usually mapped on the correct unit. Even if it is mapped onto a nearby unit, the associative connections are likely to point to the correct associative unit.

In some cases a noisy input representation may be closest to the correct word representation, but even closer to the average of two representations. In this case it might be mapped on an intermediate unit which represents the average between two representations (figure 9.9), and there is only 50% chance that this unit will activate the correct association. If the associative activity is propagated through larger neighborhoods than only the maximally responding unit, even most of these cases can be corrected. Units around the correct representation generate stronger responses than the units around the competing representation, and the total pattern of strong responses is centered around the correct unit.

In DISCERN, the activity is propagated through the associative connections of the maximally responding unit and *the three most active units in its immediate neighborhood*:

$$\eta_{O,kh} = \sigma \left(\sum_{(i,j) \in N_4} a_{ij,kh} \eta_{I,ij} \right) \quad (9.3)$$

where $a_{ij,kh}$ is the weight between the input map unit (i, j) (either lexical or semantic) and the (associative) output map unit (k, h) (semantic or lexical), and $\eta_{I,ij}$ and $\eta_{O,kl}$ indicate the activities of these units (see function `translate` in appendix D.1.4). N_4 stands for the set of the four most active units in the neighborhood. Usually these units reflect the first moment (i.e. center of gravity) of the activity better than the image unit alone, and they are most strongly connected to the correct unit in the associative map. As a result, the correct associative unit receives the highest total activity and the correct, noise-free output representation can be recovered from the weights of this unit.

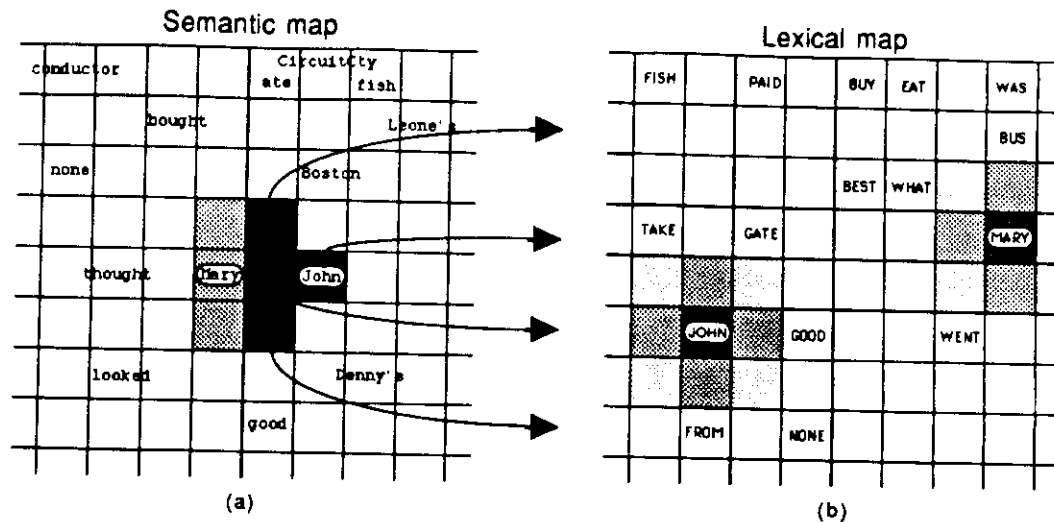


Figure 9.9: **Correcting a noisy input representation.** When a noisy John is presented to the semantic map, the intermediate unit between John and Mary generates the strongest response. The activity propagates through this unit and also three most active neighboring units, i.e. above, below and on the right. In the lexical map, the correct unit JOHN receives the largest total activity. Only part of each map is shown.

Figures 9.10 and 9.11 plot the correctness and accuracy of the image unit representation in the lexical and semantic input maps, and the correctness and accuracy of the associated representation, as the input vector becomes increasingly noisy.

The lexicon filters out noise very efficiently. The semantic and the semantic → lexical representations are almost completely accurate with up to 35% noise. With more noise, the accuracy deteriorates quickly as input vectors become incorrectly classified. The lexical and lexical → semantic representations show similar behaviour, although the classification breaks down sooner, at around 15% noise. This is because the lexical representations are less distributed. Each letter is represented by a single unit, and a change of a single letter may change the whole word. This makes the lexical representations more sensitive to noise than the semantic representations. Note that the associative mappings are actually more accurate than the input map representations in both directions.

The graphs show that the lexicon is performing highly categorical processing, similar to the episodic memory maps (see figure 7.10). As long as the input representation is closer to the correct representation than its competitors, it is replaced by the noise-free representation stored in the feature maps.

9.5 Modeling the human lexical system

The lexicon in DISCERN contains a single lexical modality (orthographic), and the same representation space is used for both input and output. This is a practical design for an AI module, and illustrates the basic principles and properties of the approach. A more plausible model of the human lexical system would contain both the orthographic and phonological

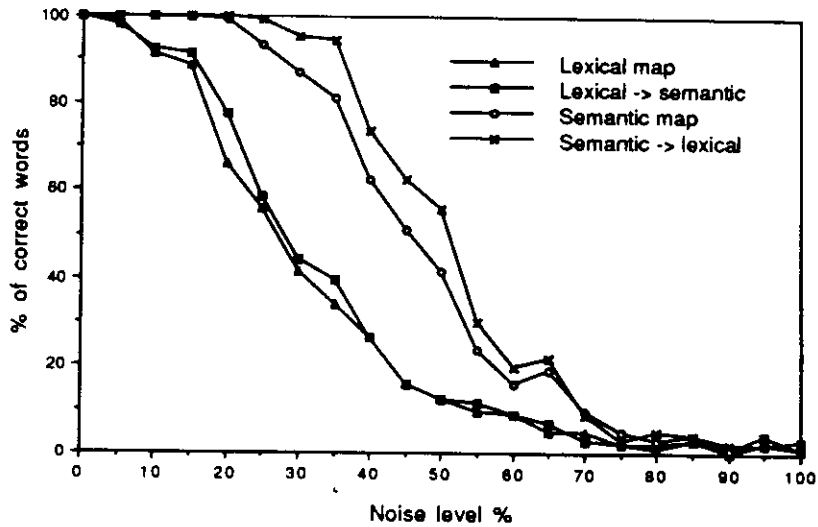


Figure 9.10: Number of correct words with noisy input. The input vector components were obtained from $c_i = (1 - p)r_i + p * X$, where r_i is the story representation component, p is the noise level, and X is a random variable uniformly distributed within $[0,1]$.

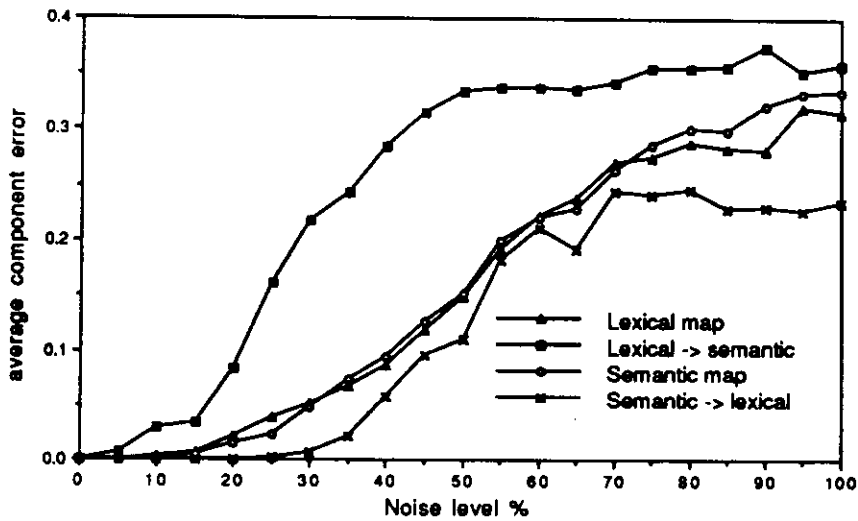


Figure 9.11: Accuracy of the word representations with noisy input.

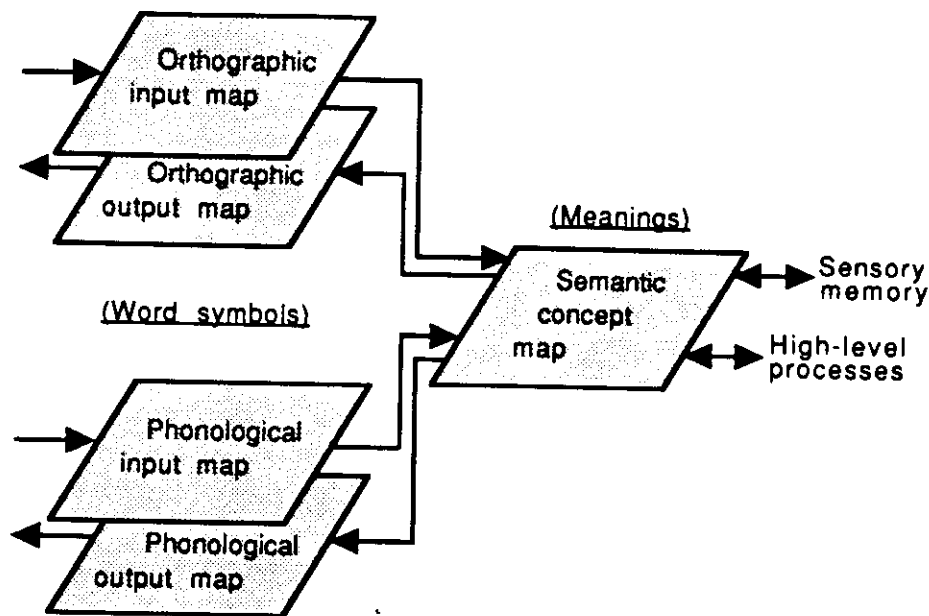


Figure 9.12: The DISLEX model of the human lexical system. The lexical symbol memories are modality and direction specific. The arrows indicate pathways of distributed representations.

modalities, and would have separate channels for input and output.

9.5.1 The DISLEX architecture

DISLEX (DISTRIBUTED feature map LEXicon) [Miikkulainen, 1990a] is an extension of the lexicon architecture in DISCERN which aims at more accurate modeling of the human lexical system. The semantic memory of DISLEX consists of distributed representations of the distinct concepts, as before. However, there are separate symbol memories for each input and output modality (figure 9.12).

These memories store distributed representations for the lexical word symbols in different modalities. For example, the orthographic word representation for DOG consists of the visual form of the letters D, O, G, and the phonological representation stands for the string of phonemes *do:g*. A central assumption in DISLEX is that the representations in each modality reflect the similarities within that modality. For example, the orthographic representations for DOG and DOC are very similar, but less so in the phonological domain.

The different symbol memories and the semantic memory are implemented as feature maps. There is one map for each input and output modality and one for the semantic memory. The maps are connected with one-way associative connections (figure 9.12). There is also a pathway from the semantic memory to the higher level language processing systems, which use the semantic representations. The semantic memory is also connected to the sensory memory, which contains visual images of objects and other sensory information. This pathway allows nonlinguistic access to the semantic memory, and provides the means for symbol grounding [Harnad, 1989]. The semantic word representation contains sensory

information about the word referent, and the abstract word meaning originating from the high-level processes. This approach fits in nicely with the ID+content technique for word representation (sections 4.5, 12.6.2 and 13.2.2).

The separation of modality-specific channels is intuitively compelling, since the modalities give rise to different representations, and are processed through different structures [Caramazza, 1988]. The symbol spaces are not identical across modalities; there are homophones and homographs. Considerable experimental evidence also supports dissociation of the lexical components [Caramazza, 1988]. Some of this is outlined below.

9.5.2 Modeling aphasia

The DISLEX architecture is in good agreement with the current theories of the human lexical system, such as those of [Caramazza, 1988; Warrington and McCarthy, 1987; Warrington, 1975]. Many observed lexical deficits in acquired aphasia have straightforward explanations in the model.

A common feature of the aphasic deficits is category specificity. The patient may have difficulties only with words belonging to a specific syntactic or semantic category. In certain patients the lexical access to e.g. function words is selectively impaired, in other cases the patient has trouble with verbs [Caramazza, 1988; Coltheart et al., 1980]. More specific impairments seem to occur in semantic hierarchies. Some patients have trouble with e.g. concrete words, or inanimate objects [Warrington and Shallice, 1984], or even as specific classes as names of fruits and vegetables [Hart et al., 1985].

Deficits of this kind can be explained by the topological organization of the semantic memory. The semantic map in DISLEX is hierarchically organized, and reflects both the syntactic and semantic properties of the words. Localized lesions to the map produce selective impairments, like the above.

In some cases the impairments cover all modalities, sometimes they are limited only to verbal input or output, or even only to orthographic or phonological domain. This suggests that the semantic memory, visual input, and verbal input/output modalities are represented in separate structures, strongly supporting the distributed DISLEX architecture.

For example, some patients were unable to access the specific meanings from verbally as well as visually (with pictures) presented cues [Warrington, 1975; Warrington and Shallice, 1984]. This implies that the semantic memory itself, i.e. the map, had been damaged. Another patient could not give definitions for aurally presented names of living things such as "dolphin", although he was able to describe other objects. But when shown a picture of a dolphin, he could name it and give an accurate verbal description of it [McCarthy and Warrington, 1988]. This suggests that the visual pathway to the semantic memory, the semantic memory itself, and the verbal output were preserved, but the verbal access to the semantic memory had been damaged. In another case, the patient was unable to name fruits and vegetables, although he was able to match their names with pictures, and classify them correctly when their names were presented aurally [Hart et al., 1985]. In other words, his semantic memory and verbal input were preserved, and the verbal output function was selectively impaired.

The impairment of semantic categories, which is restricted to a single input or output modality, can be explained in DISLEX by severed pathways between lexical and semantic maps. The pathways are not single axons, but consist of interneurons, which also exhibit map-like organization. Close to the semantic map, the organization is semantic, close to the lexical map it parallels the lexical map. If the pathway is severed close to the semantic map, semantic impairment within this modality results.

The dissociation of the orthographic and phonological modalities is also well-documented. Some patients have deficits only in one of the input or output channels, or different deficits in different channels [Basso et al., 1978]. For example, a patient may have spelling difficulties exclusively in the orthographic output domain [Goodman and Caramazza, 1986; Miceli et al., 1985]. The types of errors in visual and phonological dyslexia (section 9.3.6) further indicate that the channels are organized according to the physical forms of the words. The DISLEX model predicts that it would be possible to lose access to specific types of symbols, as a result of localized damage to a lexical map.

In the aphasic impairments, the high-frequency words are often better preserved than rare words. This is also predicted by the feature map organization. The most often occurring words occupy larger areas in the map, making them more robust against damage.

Part IV
EVALUATION

Chapter 10

Behaviour of the complete model

In the previous chapters, the components of DISCERN have been presented and their operation and performance have been analyzed, at times with data that better illustrates their properties than the script paraphrasing data of DISCERN. In this chapter, all the components are brought together, and the performance of the complete DISCERN model is analyzed. With traces of the actual output of DISCERN, the behaviour of the model is discussed in high-level terms and compared to human performance.

10.1 Connecting the modules

As has been pointed out in previous chapters, the DISCERN modules can be trained simultaneously, using compatible training data and the same set of word representations. This way the mappings are likely to be smoother, i.e. exhibit more continuity between image units, because the representations have evolved over time. In practice it is more efficient to train the processing modules first, and organize the lexicon and episodic memories only after the final set of semantic representations has been established. This is how DISCERN was trained (the training programs and data are listed in appendix C).

For the performance phase, the processing modules trained in chapter 5, the hierarchical feature maps presented in chapter 7 and the lexicon of section 9.4 were brought together (the performance code of DISCERN is listed in appendix D). The test data for the complete system consists of the same 96 story instantiations and questions as in chapter 5 (samples in appendix D.3). However, nonextreme ID values (0.75,0.25) and (0.25,0.75) were used for the two instances instead of the extrem values (1,0) and (0,1). The nonextreme values are better suited to the properties of feature maps, whereas the FGREP modules should have no trouble keeping them separate.

Performance figures for each module in isolation are listed in tables 10.1 and 10.2. The networks were not connected, but the input patterns for each network were formed from compatible script-processing data. The performance with this data turns out to be almost exactly the same as with the ideal data sets, used in the performance tests for the different modules in the previous chapters. The processing modules are just as accurate with non-extreme ID values as with extreme IDs. The actual word frequency gives the same results for the lexicon components as uniformly distributed word data. The figures for the hierarchical feature maps in storing (table 10.1) and retrieving (table 10.2) are exactly the same, indicating that each stored representation is successfully retrieved with a perfect cue.

Tables 10.3 and 10.4 give the same performance figures when the modules are connected in a chain, the output of one module directly feeding the input of the next module. The input for the story generator was obtained from the episodic memory representation directly,

Module	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical input map	100	100	100	.0000
Semantic output map	100	100	100	.0002
Sentence parser	99	99	100	.0123
Story parser	100	100	100	.0091
Hier.feature maps	99	99	99	.0055
Story gener	100	100	100	.0082
Sentence gener	100	100	100	.0202
Semantic input map	100	100	100	.0002
Lexical output map	100	100	100	.0000

Table 10.1: **Performance of isolated DISCERN modules in paraphrasing.** The data consists of 96 instantiations of the full story skeletons in appendix A, with two instances for each prototype word. The first column indicates the percentage of correct words out of all output words, the second shows the percentage for word instances created with the ID+content technique. The third column indicates the percentage of output units which were within 0.15 of the correct value, and the last column shows the average error per output unit. The numbers for the hierarchical feature maps indicate the accuracy of the *stored representations*, obtained from the weights of the image units as the representations are stored in the episodic memory.

Module	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical input map	100	100	100	.0000
Semantic output map	100	100	100	.0002
Sentence parser	99	99	99	.0168
Cue former	82	82	89	.0385
Hier.feature maps	99	99	99	.0056
Answer producer	100	100	100	.0066
Sentence gener	100	100	100	.0191
Semantic input map	100	100	100	.0002
Lexical output map	100	100	100	.0000

Table 10.2: **Performance of isolated DISCERN modules in question answering.** The figures for the hierarchical feature maps indicate the accuracy of the *retrieved representations*, when an exact representation was used as a cue.

Module	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical input map	100	100	100	.0000
Semantic output map	100	100	100	.0002
Sentence parser	99	99	100	.0123
Story parser	99	99	99	.0139
Hier.feature maps	98	94	99	.0064
Story gener	99	97	99	.0106
Sentence gener	99	96	99	.0260
Semantic input map	98	93	99	.0030
Lexical output map	99	96	100	.0024

Table 10.3: Paraphrasing performance of connected DISCERN modules.

Module	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical input map	100	100	100	.0000
Semantic output map	100	100	100	.0002
Sentence parser	99	99	99	.0168
Cue former	82	81	89	.0464
Hier.feature maps	98	94	99	.0065
Answer producer	99	96	99	.0128
Sentence gener	98	96	99	.0285
Semantic input map	97	91	99	.0045
Lexical output map	98	95	99	.0039

Table 10.4: Question answering performance of connected DISCERN modules.

without cueing the memory. Tables 10.5 and 10.6 were obtained otherwise with the same data, but the input stories were incomplete, each consisting of only three sentences (see the description of data in appendix A).

The modules perform quite well together. About 98% of the output words are correct, which is rather remarkable for a chain 9 modules long, consisting of several different types of modules. None of the modules is completely accurate, and each one introduces errors to the flow of information. The errors accumulate remarkably little in the chain. The modules communicate with representations which are redundant, and at each module interface, noise is efficiently filtered out.

When a noisy concept representation is input to an FGREP module, the small deviations in the components cancel out. If the noise is uniformly distributed, the network can process noisy patterns almost as well as clean ones. In the feature maps, as long as the noisy pattern is closest to the correct one, it is mapped on the correct unit, and the noise is completely removed.

However, noise in the ID patterns cannot be removed very well. The processing networks

Module	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical input map	100	100	100	.0000
Semantic output map	100	100	100	.0002
Sentence parser	99	99	100	.0129
Story parser	94	91	94	.0353
Hier.feature maps	93	89	96	.0153
Story gener	98	92	99	.0130
Sentence gener	98	93	99	.0289
Semantic input map	95	83	98	.0065
Lexical output map	98	93	99	.0046

Table 10.5: Paraphrasing performance of connected DISCERN modules, incomplete input stories.

Module	All words	Instances	$E_i < 0.15$	E_{avg}
Lexical input map	100	100	100	.0000
Semantic output map	100	100	100	.0002
Sentence parser	99	99	99	.0168
Cue former	82	81	89	.0464
Hier.feature maps	93	88	96	.0160
Answer producer	97	91	98	.0167
Sentence gener	97	92	97	.0339
Semantic input map	92	79	97	.0099
Lexical output map	97	92	99	.0066

Table 10.6: Question answering performance of connected DISCERN modules, incomplete input stories.

are trained to process any ID pattern as it is, and they cannot remove noise in the IDs. The only exception is the story parser, which sometimes gets to see the same instance several times, e.g. *John* in several sentences. If the noise in the ID units is random, the story parser filters a lot of it out. The feature maps in the lexicon and in the episodic memory recognize the IDs categorically, and filter out the inaccuracies quite efficiently. However, if the ID pattern has become so corrupted that it is actually closer to another ID pattern, the feature maps will make it an even better instance of the wrong ID pattern. The problem is that the IDs are not redundant, and there is no top-down information available to keep them straight. Once an ID is confused with another one, it cannot be corrected.

As the performance tables indicate, most of the errors DISCERN makes are of this type. For example, in parsing incomplete stories, the story parser introduces a number of incorrect word instances, which cannot be corrected later. The performance gets better later in the chain, but for a different reason. The output of the story parser has one slot for each instance, whereas the instances occur with different frequencies in later modules, reflecting the frequencies of these words in the stories. The main actor is repeated several times in each story, whereas e.g. the name of the restaurant is mentioned at most twice. The more frequent instances are learned better, making the performance better in later modules.

This behaviour is not entirely implausible. It means that DISCERN is more likely to confuse instances than types. It is more likely to make role binding errors that are plausible, i.e. produce role bindings which are possible in general but incorrect in this particular case. It almost never confuses words with sharply different meanings. Moreover, once the role binding error has been made, it is usually consistent throughout the paraphrasing and question answering.

The semantic map at the output often maps cloned representations incorrectly, but these errors are cleared in the semantic \rightarrow lexical mapping. For example, in answering questions about incomplete stories, only 79% of the words on the semantic map are correct, but the accuracy is back to 92% on the lexical output map (table 10.6).

In the semantic map, there are almost always one or more intermediate units between the two instances of the same word prototype, e.g. *John* and *Mary*. The intermediate unit represents the average of the two instances, i.e. has the ID values close to 0.5. The input representation produced by the sentence generator is often noisy enough so that even when it is closer to the correct instance than the wrong instance, it is even closer to the average representation (a typical such case was illustrated in figure 9.9). In the statistics, the intermediate unit is counted as either one or the other instance, and the performance figures for the word instances in the semantic output map are not good. However, the classification is corrected in the semantic \rightarrow lexical propagation. The lexical output words are just as often correct as the output of the sentence parser. And when they are correct, they are *exactly* correct. The accuracy of the lexical symbol output is much better than the accuracy of the sentence producer.

The lexicon is completely modular and does not use any top down information. Unfortunately this means that the lexicon cannot correct a semantic representation which is closer to the wrong instance. It will simply output a lexical representation which is the wrong instance *exactly*, rather than some undefined intermediate representation. We feel that the

lexicon is a highly categorical component of the human language faculty, and this is a plausible way for the lexicon to err under inaccurate circumstances.

10.2 Example run

In this section an annotated example run of DISCERN is presented. DISCERN read its input stories and questions from three input files (listed in appendix D.3.4), and printed out log information describing the I/O of each module. The activity patterns at each I/O assembly were compared to the representations in the lexicon, and the word label of the closest match was output.

DISCERN output is shown in courier typeface. Comments printed out by DISCERN are enclosed in brackets, otherwise all words stand for activity patterns, with “-” indicating a blank pattern. Output layers consisting of several assemblies are enclosed between bars. The lexicon translation is indicated by “>”.

10.2.1 Parsing

DISCERN reads the first story in the input file:

```
[ parsing input story: ]
JOHN>John WENT>went TO>to DENNY'S>Denny's .>.
|John went - - - Denny's|
JOHN>John ASKED>asked THE>the WAITER>waiter FOR>for FISH>fish .>.
|John asked waiter - fish _|
JOHN>John ATE>ate A>a BAD>bad FISH>fish .>.
|John ate - bad fish _|

[ into internal rep: ]
|$restaurant $coffee John fish Denny's bad small|

[ storing into episodic memory: ]
[ image units (0,0), (1,1), (3,6): ]
|$restaurant $coffee John fish Denny's bad small|
```

The story is an instance of the coffee-shop track of the restaurant script. The actual input to DISCERN consists of distributed representations for the lexical word symbols. Each representation is mapped on the lexical map, and the label of the image unit is output. The first word of the first story is JOHN, which is mapped on the unit whose input weights are closest to the representation of JOHN.

The activity is propagated through the associative connections to the semantic map, where the unit whose input weight vector is closest to the representation of John receives

the highest activity. The activity is propagated through the input weights of this unit to the single input assembly of the sentence parser.

The same process takes place for each word of the first sentence, until a period is input. At this point, the sentence parser propagates the activity pattern at its output layer. |John went - - - Denny's| to the input layer of the sentence parser.

The same process is repeated for each sentence in the story. After the last sentence, the activity pattern at the output layer of the story parser network, |\$restaurant \$coffee John fish Denny's bad small| constitutes the internal representation of the story.

This representation is presented to the hierarchical feature map system, which classifies it as an instance of the restaurant script (map (0,0)) and coffee-shop track (submap (1,1)), and maps it on unit (3,6) on this map. The combined weights of the image units at the three levels of the hierarchy give the representation of the story in the episodic memory. |\$restaurant \$coffee John fish Denny's bad small|.

10.2.2 Paraphrasing

The episodic memory representation is fed to the input layer of the story generator network, and DISCERN now generates a fully expanded paraphrase of the story:

```
[ generating paraphrase: ]
|John went - - - Denny's|
John>John>JOHN went>went>WENT to>to>TO Denny's>Denny's>DENNY'S .>.>.
|John seated - - John _|
John>John>JOHN seated>seated>SEATED John>John>JOHN .>.>.
|John asked waiter - fish _|
John>John>JOHN asked>asked>ASKED the>the>THE waiter>waiter>WAITER
for>for>FOR fish>fish>FISH .>.>.
|John ate - bad fish _|
John>John>JOHN ate>ate>ATE a>a>A bad>bad>BAD fish>fish>FISH .>.>.
|John left waiter small tip _|
John>John>JOHN left>left>LEFT a>a>A small>small>SMALL tip>tip>TIP .>.>.
|John paid cashier - - _|
John>John>JOHN paid>paid>PAID the>the>THE cashier>cashier>CASHIER .>.>.
|John left - - - Denny's|
John>John>JOHN left>left>LEFT Denny's>Denny's>DENNY'S .>.>.
```

The story generator produces the case-role representation of the first sentence, |John went - - - Denny's| as its first output pattern. This propagates to the input of the sentence generator, which outputs the semantic representations of the words one at a time. Each word, e.g. the first word John, is presented to the semantic map of the lexicon, where it is mapped on the unit whose input weights are closest to the representation of John. The

activity propagates through the associative weights to the lexical map, where the unit JOHN lights up strongest. The representation stored in the input weights of this unit is output as the first word in the stream of lexical output word representations. The same process repeats for each sentence and word in the story.

A few high-level observations can be made on the paraphrase. At the high-level, DISCERN has correctly inferred a number of events which were not mentioned in the original story, and inserted them in the paraphrase. In the coffee-shop stories the quality of food correlates with the amount of tip. Consequently, DISCERN has also inferred that because the fish was bad, John must have left a small tip.

10.2.3 Basic question answering

For the sake of legibility, the semantic mapping and sentence parser output log will be disabled from now on, except for a few examples in later sections which illustrate lexical error behaviour and clean-up. DISCERN now goes on to read four more stories:

```
[ parsing input story: ]
MARY WENT TO LEONE'S .
|Mary went - - - Leone's|
MARY ASKED THE WAITER FOR STEAK .
|Mary asked waiter - steak -|
MARY LEFT A BIG TIP .
|Mary left waiter big tip -|
```

```
[ into internal rep: ]
|$restaurant $fancy Mary steak Leone's good big|
```

```
[ storing into episodic memory: ]
[ image units (0,0), (0,0), (7,1): ]
|$restaurant $fancy Mary steak Leone's good big|
```

```
[ parsing input story: ]
MARY WENT TO CIRCUITCTY .
|Mary went - - - CircuitCty|
MARY LOOKED FOR GOOD TV .
|Mary looked - good tv -|
MARY TOOK THE BEST TV .
|Mary took - best tv -|
```

```
[ into internal rep: ]
|$shopping $electronics Mary tv CircuitCty - -|
```

```
[ storing into episodic memory: ]
[ image units (1,0), (1,0), (7,1): ]
```


!\$shopping \$electronics Mary tv CircuitCty - -|

[parsing input story:]
JOHN WENT TO RADIOSHACK .
|John went - - - RadioShack|
JOHN LOOKED FOR GOOD TV .
|John looked - good tv -|
JOHN TOOK THE BEST TV .
|John took - best tv -|

[into internal rep:]
!\$shopping \$electronics John tv RadioShack - -|

[storing into episodic memory:]
[image units (1,0), (1,0), (1,6):]
!\$shopping \$electronics John tv RadioShack - -|

[parsing input story:]
MARY WENT TO DFW .
|Mary went - - - dfw|
MARY CHECKED-IN FOR A FLIGHT TO SFO .
|Mary checked-in - sfo flight -|
THE PLANE ARRIVED AT SFO .
|plane arrived - - - sfo|

[into internal rep:]
!\$travel \$plane Mary dfw sfo big -|

[storing into episodic memory:]
[image units (1,1), (1,0), (1,0):]
!\$travel \$plane Mary dfw sfo big -|

DISCERN has now read five stories: two instances of electronics-shopping and one of each of coffee-shop-restaurant, fancy-restaurant and plane-travel. It is now asked a number of questions about the contents of the stories. The questions are presented in random order. DISCERN first has to determine which story the question is referring to, and then formulate an answer. Let us look at the processing of the first question in detail:

[parsing question:]
WHAT DID MARY EAT AT LEONE'S ?
|Mary ate - - what Leone's|

[into cue:]
!\$restaurant \$fancy Mary steak Leone's good *from(big)*|

```
[ retrieving from episodic memory: ]
[ map (0,0), (0,0): →(4,3):0.813 →(7,1):0.985 →(7,1):0.997
                    →(7,1):0.998 →(7,1):0.999 →(7,1):0.999
                    →(7,1):0.999 →(7,1):0.999 →(7,1):0.999 ]
[ image units (0,0), (0,0), (7,1): ]
|$restaurant $fancy Mary steak Leone's good big|
```

```
[ generating answer: ]
|Mary ate - good steak -|
MARY ATE A GOOD STEAK .
```

The sentence parser reads the question sentence as usual, and forms a case role representation of it, *Mary ate - - what Leone's*. This pattern is sent to the cue former, which makes a best guess about the story that this question is referring to. A number of role bindings can be safely filled in, but a few of them are unspecified and the cue former generates a pattern which is an average of the different alternatives. Above, the average of *lobster* and *steak* happens to be closer to *steak* (numerically). The average of *big* and *small* happens to be closest to *from* in the lexicon, and this word appears in the transcript above. Incorrect words in the examples are enclosed between asterisks, together with the correct word in parenthesis).

The cue is presented to the episodic memory, which classifies it as an instance of the restaurant script (0,0) and fancy-restaurant track (0,0). The initial response of the fancy-restaurant role-binding map is centered around the unit (4,3) with activity 0.813, but quickly settles to (7,1), where the trace was stored. The story representation is obtained by combining the weights of the three image units.

The answer producer takes the cue representation and the story representation as its input and generates the case-role representation of the answer, *|Mary ate - good steak -|*. The sentence generator produces words of the answer sentence one at a time, and the lexicon translates each one of them into the corresponding lexical symbol.

In a similar manner, DISCERN processes another question about the same story:

```
[ parsing question: ]
HOW DID MARY LIKE STEAK AT LEONE'S ?
|Mary thought - what steak Leone's|
```

```
[ into cue: ]
|$restaurant $fancy Mary steak Leone's good *from(big)*|
```

```
[ retrieving from episodic memory: ]
[ map (0,0), (0,0): →(4,3):0.813 →(7,1):0.987 →(7,1):0.998
                    →(7,1):0.998 →(7,1):0.999 →(7,1):0.999
                    →(7,1):0.999 →(7,1):0.999 →(7,1):0.999 ]
```

[image units (0,0), (0,0), (7,1):]
|\$restaurant \$fancy Mary steak Leone's good big|

[generating answer:]
|Mary thought - good steak Leone's|
MARY THOUGHT STEAK WAS GOOD AT LEONE'S .

The two questions above refer to events and role fillers which *were not mentioned* in the original story. DISCERN was told that Mary went to Leone's and ordered a steak, but nothing was said about eating the steak or about the quality of the food. This makes no difference to DISCERN. It read the story, immediately made the appropriate inferences, and can no longer tell what was mentioned and what was inferred. Eating is always part of the restaurant visit, and it knows that *steak* was the food in this visit. It confidently declares that Mary ate the steak. Moreover, it believes that the steak was good, because food is always good in fancy-restaurants (in our data). A similar situation occurs in the context of the plane-travel story:

[parsing question:]
WHERE DID MARY TAKE A PLANE TO ?
|Mary took - - plane where|

[into cue:]
|\$travel \$plane Mary *lax(dfw)* sfo big -|

[retrieving from episodic memory:]
[map (1,1), (1,0): →(4,3):0.672 →(1,0):0.843 →(1,0):0.963
→(1,0):0.955 →(1,0):0.980 →(1,0):0.968
→(1,0):0.984 →(1,0):0.974 →(1,0):0.986]

[image units (1,1), (1,0), (1,0):]
|\$travel \$plane Mary dfw sfo big -|

[generating answer:]
|Mary took - - plane sfo|
MARY TOOK A PLANE TO SFO .

[parsing question:]
DID MARY TRAVEL BIG DISTANCE TO SFO ?
|Mary traveled - big distance sfo|

[into cue:]
|\$travel \$plane Mary *lax(dfw)* sfo big -|

```
[ retrieving from episodic memory: ]
[ map (1,1), (1,0): →(2,3):0.786 →(1,0):0.607 →(1,0):0.912
      →(1,0):0.784 →(1,0):0.944 →(1,0):0.830
      →(1,0):0.954 →(1,0):0.847 →(1,0):0.957 ]
[ image units (1,1), (1,0), (1,0): ]
|$travel $plane Mary dfw sfo big _|
```

```
[ generating answer: ]
|Mary traveled _ big distance _|
MARY TRAVELED A BIG DISTANCE .
```

DISCERN assumes that after checking in, Mary actually flew to SFO. The distance traveled was not mentioned in the story, but it is always big in plane-travel.

10.2.4 Disambiguating the question

```
[ parsing question: ]
WHERE DID JOHN EAT FISH ?
|John ate _ _ fish where|
```

```
[ into cue: ]
|$restaurant $coffee John fish Denny's bad *from(small)*|
```

```
[ retrieving from episodic memory: ]
[ map (0,0), (1,1): →(4,3):0.804 →(3,6):0.993 →(3,6):0.998
      →(3,6):0.998 →(3,6):0.999 →(3,6):0.999
      →(3,6):0.999 →(3,6):0.999 →(3,6):0.999 ]
[ image units (0,0), (1,1), (3,6): ]
|$restaurant $coffee John fish Denny's bad small|
```

```
[ generating answer: ]
|John ate _ _ fish Denny's|
JOHN ATE FISH AT DENNY'S .
```

This example illustrates how DISCERN uses the information provided by the question to come up with a guess for the target story. A question about eating must refer to a restaurant story, but which track? If the restaurant was mentioned, it would be easy to tell whether it is a fancy, coffee-shop or fast-food restaurant. However, the food is just as good a cue. Fish is only eaten in coffee-shop restaurants in our data, and DISCERN generates a cue for a coffee-shop story with John as the customer and fish as food. The rest of the role bindings are guesses, averages of the different alternatives. The cue is easily mapped to the coffee-shop role-binding map, which completes the guesses by settling onto an existing trace.

DISCERN has read two different electronics-shopping stories, and they are stored on the same map. In situations like this, the retrieval process has to disambiguate between the alternative traces:

[parsing question:]

WHERE DID MARY BUY TV ?

|Mary bought - - tv where|

[into cue:]

|\$shopping \$electronics Mary tv CircuitCty - -|

[retrieving from episodic memory:]

[map (1,0), (1,0): →(7,1):0.767 →(7,1):0.981 →(7,1):0.990
→(7,1):0.990 →(7,1):0.993 →(7,1):0.994
→(7,1):0.995 →(7,1):0.995 →(7,1):0.995]

[image units (1,0), (1,0), (7,1):]

|\$shopping \$electronics Mary tv CircuitCty - -|

[generating answer:]

|Mary bought - - tv CircuitCty|

MARY BOUGHT TV AT CIRCUITCTY .

[parsing question:]

WHERE DID JOHN BUY TV ?

|John bought - - tv where|

[into cue:]

|\$shopping \$electronics John tv RadioShack - -|

[retrieving from episodic memory:]

[map (1,0), (1,0): →(1,6):0.777 →(1,6):0.984 →(1,6):0.993
→(1,6):0.993 →(1,6):0.994 →(1,6):0.995
→(1,6):0.995 →(1,6):0.994 →(1,6):0.994]

[image units (1,0), (1,0), (1,6):]

|\$shopping \$electronics John tv RadioShack - -|

[generating answer:]

|John bought - - tv RadioShack

JOHN BOUGHT TV AT RADIOSHACK .

Both of these questions are classified as referring to an electronics-shopping story. The customer is specified in both questions, and the filler of this role is different in the two traces. The initial response of the electronics-shopping map is closer to the trace with the matching customer, and the activity settles at around this trace. In other words, the story which better matches the question is retrieved.

10.2.5 Ambiguous questions

The above examples were easy for DISCERN in that the correct role-binding map could be unambiguously determined from the question. In the following examples, the cue is deliberately incomplete and ambiguous, and DISCERN has to look at several maps to find the appropriate trace. The search threshold was increased from 1.0 to 3.0 to allow DISCERN to look at maps which do not match as well with the cue.

```
[ parsing question: ]
WHO ATE ?
|who ate - - *(fish)* Denny's|

[ into cue: ]
|$restaurant $coffee *Mary(John)* *spaghetti(fish)* Denny's *(bad)*
from(small)*|

[ retrieving from episodic memory: ]
[ map (0,0), (0,0): →(3,3):0.813 →(7,1):0.980 →(7,1):0.997
                    →(7,1):0.997 →(7,1):0.998 →(7,1):0.998
                    →(7,1):0.999 →(7,1):0.999 →(7,1):0.999 ]
[ map (0,0), (0,1): →(1,0):0.767 →(1,0):0.000 →(1,0):0.767
                    →(1,0):0.000 ]
[ map (0,0), (1,0): →(7,7):0.148 →(7,7):0.006 →(7,7):0.131
                    →(7,7):0.009 ]
[ map (0,0), (1,1): →(4,3):0.808 →(3,6):0.992 →(3,6):0.998
                    →(3,6):0.998 →(3,6):0.999 →(3,6):0.999
                    →(3,6):0.999 →(3,6):0.999 →(3,6):0.999 ]
[ image units (0,0), (1,1), (3,6): ]
|$restaurant $coffee John fish Denny's bad small|

[ generating answer: ]
|John did - - - |
JOHN DID .
```

This question is incompletely specified. Compared to the questions DISCERN was trained on, several words have been left out from the end. These words could normally be used to specify the track, but this information is not available. The sentence parser and the cue former produce patterns which are averages of the possible alternatives.

A question referring to eating refers to a restaurant story, but everything else is a guess. DISCERN looks at all role-binding maps under restaurant for possible traces. It finds nothing in the fast-food map (0,1) or in the unclaimed map (1,0), but both the fancy and the coffee-shop maps settle into a stored trace. The response of the fancy-restaurant map happens to be stronger in this case, and it is selected.

Either one of these stories can be reliably retrieved if more is specified in the question:

[parsing question:]

WHO ATE STEAK ?

|who ate - - steak *MaMaison(Leone's)*|

[into cue:]

|\$restaurant \$fancy Mary steak *MaMaison(Leone's)* good *from(big)*|

[retrieving from episodic memory:]

[map (0,0), (0,0): →(4,3):0.811 →(7,1):0.984 →(7,1):0.997
→(7,1):0.997 →(7,1):0.999 →(7,1):0.999
→(7,1):0.999 →(7,1):0.999 →(7,1):0.999]

[map (0,0), (1,0): →(7,7):0.148 →(7,7):0.006 →(7,7):0.131
→(7,7):0.009]

[map (0,0), (1,1): →(5,3):0.779 →(3,6):0.960 →(3,6):0.993
→(3,6):0.989 →(3,6):0.994 →(3,6):0.991
→(3,6):0.995 →(3,6):0.992 →(3,6):0.995]

[image units (0,0), (0,0), (7,1):]

|\$restaurant \$fancy Mary steak Leone's good big|

[generating answer:]

|Mary did - - - -|

MARY DID .

[parsing question:]

WHAT DID JOHN EAT ?

|John ate - - what Denny's|

[into cue:]

|\$restaurant \$coffee John fish Denny's *(bad)* *from(small)*|

[retrieving from episodic memory:]

[map (0,0), (0,0): →(6,3):0.762 →(7,1):0.973 →(7,1):0.996
→(7,1):0.996 →(7,1):0.998 →(7,1):0.998
→(7,1):0.999 →(7,1):0.999 →(7,1):0.999]

[map (0,0), (0,1): →(1,6):0.781 →(1,6):0.000 →(1,6):0.781
→(1,6):0.000]

[map (0,0), (1,0): →(7,7):0.148 →(7,7):0.006 →(7,7):0.13
→(7,7):0.009]

[map (0,0), (1,1): →(4,3):0.805 →(3,6):0.993 →(3,6):0.999
→(3,6):0.998 →(3,6):0.999 →(3,6):0.999
→(3,6):0.999 →(3,6):0.999 →(3,6):0.999]

[image units (0,0), (1,1), (3,6):]

|\$restaurant \$coffee John fish Denny's bad small|

```
[ generating answer: ]
|John ate - bad fish -|
JOHN ATE A BAD FISH .
```

The first question reveals that food was of the fancy type. The cue former can now generate an accurate representation for the script, track, R/food and R/taste. The correct trace creates an even stronger response than before, and is selected.

In the second question, the customer is specified. This does not specify the track, but it is enough to disambiguate between the two traces, because they have different customers. The fancy trace with customer=Mary responds weaker and the coffee-shop trace with customer=John responds stronger than before, and the coffee-shop story is selected.

In the following example, there are again two traces competing for the same question, but now the traces are located on the same role-binding map:

```
[ parsing question: ]
WHO BOUGHT TV ?
|who bought - - tv RadioShack|
```

```
[ into cue: ]
|$shopping $electronics John tv *CircuitCty(RadioShack)* - -|
```

```
[ retrieving from episodic memory: ]
[ map (1,0), (0,0): →(5,1):0.781 →(5,1):0.000 →(5,1):0.781
                    →(5,1):0.000 ]
[ map (1,0), (0,1): →(6,0):0.772 →(6,0):0.000 →(6,0):0.772
                    →(6,0):0.000 ]
[ map (1,0), (1,0): →(6,4):0.764 →(1,6):0.973 →(1,6):0.990
                    →(1,6):0.990 →(1,6):0.993 →(1,6):0.993
                    →(1,6):0.994 →(1,6):0.994 →(1,6):0.994 ]
[ map (1,0), (1,1): →(7,7):0.148 →(7,7):0.006 →(7,7):0.131
                    →(7,7):0.009 ]
[ image units (1,0), (1,0), (1,6): ]
|$shopping $electronics John tv RadioShack - -|
```

```
[ generating answer: ]
|John did - - - -|
JOHN DID .
```

In the first of the two electronics-shopping stories DISCERN has read, Mary bought a TV at CircuitCty, while in the second one, John bought one at RadioShack. The question is completely ambiguous regarding the target story. However, the traces are located on the same area of the map, and the later trace has stolen units from the earlier one. It has a

wider basin of support in that general area, and an ambiguous cue is drawn to the center of the later trace. As a result, John is returned as the customer.

The older trace still exist, but to get to it, it is necessary to specify it more carefully in the question:

```
[ parsing question: ]
WHO BOUGHT TV AT CIRCUITCTY ?
|who bought - - tv CircuitCty|

[ into cue: ]
|$shopping $electronics *John(Mary)* tv CircuitCty - -|

[ retrieving from episodic memory: ]
[ map (1,0), (0,0): →(1,2):0.781 →(1,2):0.000 →(1,2):0.781
→(1,2):0.000 ]
[ map (1,0), (0,1): →(1,7):0.784 →(1,7):0.000 →(1,7):0.784
→(1,7):0.000 ]
[ map (1,0), (1,0): →(0,5):0.785 →(7,1):0.972 →(7,1):0.985
→(7,1):0.983 →(7,1):0.988 →(7,1):0.989
→(7,1):0.990 →(7,1):0.990 →(7,1):0.990 ]
[ map (1,0), (1,1): →(7,7):0.148 →(7,7):0.006 →(7,7):0.131
→(7,7):0.009 ]
[ image units (1,0), (1,0), (7,1): ]
|$shopping $electronics Mary tv CircuitCty - -|

[ generating answer: ]
|Mary did - - - -|
MARY DID .
```

In general, incomplete questions can be used to cue DISCERN just as well as the complete ones it was trained on. The sentence parser and the cue former will use whatever information was specified in the question to come up with the best guess. The episodic memory then comes up with the story representation which best matches the cue in the memory. If the question is ambiguous between stories in the same map, the most recent one is retrieved. Ties between possible stories in different maps are broken arbitrarily.

10.2.6 Misleading and incorrect questions

DISCERN can also answer questions which provide slightly inaccurate information. The incorrect role bindings in the question are automatically corrected during retrieval:

```
[ parsing question: ]
HOW DID MARY LIKE LOBSTER AT LEONE'S ?
```

```

|Mary thought - what *lobster(steak)* Leone's|

[ into cue: ]
|$restaurant $fancy Mary *lobster(steak)* Leone's good *from(big)*|

[ retrieving from episodic memory: ]
[ map (0,0), (0,0): →(4,3):0.823 →(7,1):0.985 →(7,1):0.997
                    →(7,1):0.998 →(7,1):0.999 →(7,1):0.999
                    →(7,1):0.999 →(7,1):0.999 →(7,1):0.999 ]
[ map (0,0), (1,0): →(7,7):0.148 →(7,7):0.006 →(7,7):0.131
                    →(7,7):0.009 ]
[ map (0,0), (1,1): →(0,3):0.781 →(3,6):0.945 →(3,6):0.991
                    →(3,6):0.985 →(3,6):0.991 →(3,6):0.986
                    →(3,6):0.992 →(3,6):0.987 →(3,6):0.993 ]
[ image units (0,0), (0,0), (7,1): ]
|$restaurant $fancy Mary steak Leone's good big|

[ generating answer: ]
|Mary thought - good steak Leone's|
MARY THOUGHT STEAK WAS GOOD AT LEONE'S .

```

The question assumes that Mary ate lobster at Leone's, although she actually had a steak. The cue representation now accurately represents **lobster**. However, the only trace available in the fancy-restaurant map has **steak** as food. Otherwise this trace matches the cue very well, and the activity is drawn to the center of the trace. In the retrieved vector, **steak** is accurately represented. The fancy-food in the retrieved representation has a stronger correlation with the correct answer, and the answer producer also correctly outputs **steak**.

DISCERN simply decided that because Mary's steak meal at Leone's was the only appropriate trace in memory, that must be the target of the question, and the question itself was probably inaccurate.

This example illustrates an important point: the questions do not provide hard constraints on retrieval. There is no difference between an incomplete cue and an incorrect one. If the question as a whole is close enough to a stored trace, it will be retrieved. Some information in the question is automatically completed in the process, and some other information may be overridden.

DISCERN does recognize, though, when a question is too different from anything in the memory, and should not be corrected:

```

[ parsing question: ]
WHAT DID JOHN TAKE TO JFK ?
|John took - - what jfk|

```

```

[ into cue: ]
|$travel $plane John *dfw(lax)* jfk big -|

[ retrieving from episodic memory: ]
[ map (1,1), (0,1): →(7,7):0.148 →(7,7):0.006 →(7,7):0.131
                    →(7,7):0.009 ]
[ map (1,1), (1,0): →(6,5):0.799 →(1,0):0.212 →(6,5):0.778
                    →(1,0):0.270 ]
[ oops: no image found ]

```

There is one plane-travel story in the memory, about Mary's trip to SFO. A question about John's flight to JFK has nothing in common with the trace, other than the fact that they are classified on the same map. The initial activity pattern does not settle to the trace, and DISCERN refuses to answer the question.

10.3 Cleaning up errors

Each module in DISCERN introduces noise and errors in the flow of information. Much of the inaccuracies are automatically cleaned up at the interfaces between modules. This is important because it makes DISCERN a robust, stable system. Rather than letting the errors accumulate, eventually making it impossible for later modules to operate, the system tends to automatically come back to established, correct flow of data. In this section, different forms of clean-up are illustrated with examples.

10.3.1 Tokens by story parser

```

[ parsing input story: ]
MARY WENT TO BURGERKING .
|Mary went - - - BurgerKing|
MARY ASKED THE CASHIER FOR HAMBURGER .
|Mary asked cashier - hamburger -|
MARY PAID THE CASHIER .
|Mary paid cashier - - -|
MARY SEATED MARY .
|Mary seated - - Mary -|
MARY ATE A SMALL HAMBURGER .
|Mary ate - small hamburger -|
THE HAMBURGER TASTED BAD .
|*John(Mary)* tasted - bad hamburger -|
MARY LEFT BURGERKING .
|Mary left - - - BurgerKing|

```

```

[ into internal rep: ]
|$restaurant $fast Mary hamburger BurgerKing bad none|

[ storing into episodic memory: ]
[ image units (0,0), (0,1), (3,4): ]
|$restaurant $fast Mary hamburger BurgerKing bad none|

[ parsing question: ]
HOW DID MARY LIKE HAMBURGER AT BURGERKING ?
|Mary thought - what *fries(hamburger)* BurgerKing|

[ into cue: ]
|$restaurant $fast Mary *fries(hamburger)* BurgerKing bad none|

[ retrieving from episodic memory: ]
[ map (0,0), (0,1): →(6,0):0.672 →(3,4):0.900 →(3,4):0.977
                    →(3,4):0.973 →(3,4):0.987 →(3,4):0.981
                    →(3,4):0.990 →(3,4):0.984 →(3,4):0.991 ]
[ image units (0,0), (0,1), (3,4): ]
|$restaurant $fast Mary hamburger BurgerKing bad none|

[ generating answer: ]
|Mary thought - bad hamburger BurgerKing|
MARY THOUGHT HAMBURGER WAS BAD AT BURGERKING .

```

This example shows how some of the instance confusion errors can be filtered out. The sentence parser reads one of the actor names incorrectly, making *Mary* appear more like *John* in the case-role representation of this sentence. However, this is only one of many appearances of customer in this story, most of which are parsed correctly. The story parser sees *Mary* more often in the customer-role, and it has no trouble producing the correct pattern at its output.

Confusions between instances are hard to correct in general, because they are independent of the rest of the input pattern. However, the story parser gets to see a number of sentences. It knows that certain case-roles in different sentences represent the same thing, and these roles should be filled with the same instance. The ID patterns of all these occurrences are combined in its output, and isolated errors are automatically corrected.

Instance confusion in a question is also automatically corrected during retrieval. In the above example, the patient in the question representation is closer to *fries* instead of *hamburger*. This means only that the cue is going to be slightly less accurate than it could be. The episodic memory has no trouble finding the correct trace, and the fast-food is automatically correct in the retrieved vector.

Instance confusions elsewhere in DISCERN cannot and should not be corrected. In principle, any instance pattern is possible and should be processed as it is.

10.3.2 Semantics by episodic memory

Confusions between word types can be cleaned up by any module in DISCERN. Most of such errors occur at the output of the story parser, after it has parsed an incomplete input story. The confusions are immediately corrected in the episodic memory:

```
[ parsing input story: ]
JOHN WENT TO LEONE'S .
|John went - - - Leone's|
JOHN ASKED THE WAITER FOR LOBSTER .
|John asked waiter - lobster -|
JOHN LEFT A SMALL TIP .
|John left waiter small tip -|

[ into internal rep: ]
|$restaurant $fancy John *(lobster)* Leone's good small|

[ storing into episodic memory: ]
[ image units (0,0), (0,0), (7,7): ]
|$restaurant $fancy John lobster Leone's good small|
```

In this example, the story parser output is noisy enough so that the pattern in the R/food -slot is actually closer to the representation of "?" than *lobster*. However, the story representation as a whole can still be recognized as an instance of coffee-shop-restaurant. The weights of the coffee-shop unit in the restaurant map accurately represent the content part of *coffee-food*. In this case, the ID part in the R/food slot is still closest to the correct ID, and the combined weights of the image units accurately represent *lobster*. In other words, the implausible type in R/food is automatically corrected because the episodic memory knows that it has to be *coffee-food*. There is no other way to store a coffee-shop story in the memory.

10.3.3 Semantics by FGREP modules

In some cases the episodic memory has an inaccurate representation which must be corrected by the later modules:

```
[ parsing input story: ]
MARY WENT TO CENTRALSTA .
|Mary went - - - CentralSta|
MARY BOUGHT A TICKET TO NEWYORK .
|Mary bought - NewYork ticket -|
MARY GOT-OFF AT NEWYORK .
|Mary got-off - - - NewYork|
```

```

[ into internal rep: ]
|$travel $strain Mary CentralSta NewYork *travel(small)* _|

[ storing into episodic memory: ]
[ image units (1,1), (1,1), (2,3): ]
|$travel $strain Mary CentralSta NewYork *from(small)* _|

[ generating paraphrase: ]
|Mary went - - - CentralSta|
MARY WENT TO CENTRALSTA .
|Mary bought - NewYork ticket _|
MARY BOUGHT A TICKET TO NEWYORK .
|Mary got-on - - - train|
MARY GOT-ON THE TRAIN .
|conductor punched - - ticket _|
THE CONDUCTOR PUNCHED THE TICKET .
|Mary traveled - small distance _|
MARY TRAVELED A SMALL DISTANCE .
|Mary got-off - - - NewYork|
MARY GOT-OFF AT NEWYORK .

[ parsing question: ]
DID MARY TRAVEL BIG DISTANCE TO NEWYORK ?
|Mary traveled - big distance NewYork|

[ into cue: ]
|$travel $strain Mary CentralSta NewYork *from(small)* _|

[ retrieving from episodic memory: ]
[ map (1,1), (1,1): →(3,3):0.682 →(2,3):0.991 →(2,3):0.998
                    →(2,3):0.998 →(2,3):0.999 →(2,3):0.999
                    →(2,3):0.999 →(2,3):0.999 →(2,3):0.999 ]
[ image units (1,1), (1,1), (2,3): ]
|$travel $strain Mary CentralSta NewYork *from(small)* _|

[ generating answer: ]
|Mary traveled - small distance _|
MARY TRAVELED A SMALL DISTANCE .

```

In this story, the distance that Mary traveled by train was not specified. The story parser produces a pattern in the distance slot, which is an average of the possible alternatives, **big** and **small**. This pattern happens to be closest to the representation of **travel** in the lexicon. The episodic memory knows that the distance must be either one of the alternatives, and usually one of them would be selected as the trace is created. However, in this case the repre-

sensation is mapped on a unit which lies right next to the boundary of the area distance=**big** and distance=**small**. The map tends to continuity, and the representation of distance in this unit is shifted towards the average, and it is actually closer to the representation of from than either **big** or **small**.

The story generator and the answer producer also have learned that the only possible fillers for distance are **big** and **small**, and they automatically filter out the inaccuracies in the pattern. In this case the retrieved pattern is slightly closer to **small**, and the story generator and answer producer generate a fairly accurate representation of **small** in the distance slot.

Similar clean-up of type confusions takes place at all FGREP modules in DISCERN. The FGREP network automatically generates expectations based on the rest of the input pattern, and the expectations override the exceptional input. A drawback is that truly novel role bindings are also seen as errors, and overridden (see section 11.8).

10.3.4 Input and output representations by lexicon

Another form of clean-up is performed by the lexicon. To see what is going on, the output lexicon log is turned on momentarily during the following example, which is a question referring to John's visit to Leone's:

```
[ parsing question: ]
WHAT DID JOHN EAT AT LEONE'S ?
|John ate - - what Leone's|

[ into cue: ]
|$restaurant $fancy John *steak(lobster)* Leone's good
from(small)*|

[ retrieving from episodic memory: ]
[ map (0,0), (0,0): →(1,3):0.675 →(7,7):0.944 →(7,7):0.990
→(7,7):0.990 →(7,7):0.995 →(7,7):0.993
→(7,7):0.996 →(7,7):0.995 →(7,7):0.996 ]
[ image units (0,0), (0,0), (7,7): ]
|$restaurant $fancy John lobster Leone's good small|

[ generating answer: ]
|John ate - good lobster -|
John>*Mary(John)*>JOHN ATE A GOOD LOBSTER .
```

The sentence generator produces a slightly inaccurate representation of John. While it is closer to John than any other word in the lexicon, it is still closer to the average of John and Mary. In the semantic map of the lexicon it is mapped on the intermediate unit between the two (see figure 9.9). The input weight vector of this unit happens to be slightly closer to

Mary. However, when the activity is propagated to the lexical map, it is propagated through the four most active units in the neighborhood, and the correct unit JOHN is most highly activated in the lexical map. The representation for the lexical symbol JOHN is sent to the stream of lexical output representations. The final output is exactly correct, instead of an intermediate representation between JOHN and MARY.

Similar clean-up also occurs in the input lexicon, where the noisy lexical input pattern is translated into an exact semantic representation. As long as the associative mapping can be done correctly, DISCERN performs in noisy conditions just as well as in noise-free environments.

10.4 Error behaviour

A closer look at the ways DISCERN fails and the types of errors it produces is important for two reasons: (1) it gives us insight about the limitations of the system, (2) it gives us an idea of the plausibility of DISCERN as a model of human performance. The performance should degrade gracefully as the input becomes deficient and the memory becomes overloaded, and the errors that occur should be similar to observed error behaviour in human performance.

10.4.1 Dealing with incomplete input

What happens, for example, when the input story does not provide enough information to instantiate a role binding? Consider the following example:

```
[ parsing input story: ]
MARY WENT TO BUS-STOP .
|Mary went - - - bus-stop|
MARY WAITED FOR THE BUS .
|Mary waited - - bus -|
MARY PAID THE DRIVER .
|Mary paid driver - - -|
MARY GOT-OFF THE BUS .
|Mary got-off - - - bus|

[ into internal rep: ]
|$travel $bus Mary bus-stop beach small -|

[ storing into episodic memory: ]
[ image units (1,1), (0,0), (5,0): ]
|$travel $bus Mary bus-stop beach small -|

[ generating paraphrase: ]
|Mary went - - - bus-stop|
```



```

MARY WENT TO BUS-STOP .
|Mary waited _ _ bus _|
MARY WAITED FOR THE BUS .
|Mary got-on _ beach _ bus|
MARY GOT-ON THE BEACH BUS .
|Mary paid driver _ _ _|
MARY PAID THE DRIVER .
|bus arrived _ _ _ beach|
THE BUS ARRIVED AT BEACH .
|Mary got-off _ _ _ bus|
MARY GOT-OFF THE BUS .

```

```

[ parsing question: ]
WHERE DID MARY TAKE A BUS TO ?
|Mary took _ _ bus where|

```

```

[ into cue: ]
|$travel $bus Mary *bus-termin(bus-stop)* beach small _|

```

```

[ retrieving from episodic memory: ]
[ map (1,1), (0,0): →(7,1):0.669 →(5,0):0.899 →(5,0):0.979
                    →(5,0):0.975 →(5,0):0.988 →(5,0):0.983
                    →(5,0):0.991 →(5,0):0.986 →(5,0):0.992 ]
[ image units (1,1), (0,0), (5,0): ]
|$travel $bus Mary bus-stop beach small _|

```

```

[ generating answer: ]
|Mary took _ _ bus beach|
MARY TOOK A BUS TO BEACH .

```

The input story does not specify where Mary took the bus to. The story parser produces an output pattern in the DESTINATION-slot, which is an average of all alternatives. The content part of this pattern is an accurate representation of the content of bus-destination, but the IDs have average values. However, there is always some noise in the output, and the ID values are never exactly the average. In this case, the pattern is slightly closer to beach than downtown, the other alternative. When the story representation is presented to the episodic memory, it is mapped on one of the existing representations on the role-binding map, and the DESTINATION becomes an even better representation of beach.

DISCERN has now made a guess that the DESTINATION was beach, and it considers the matter settled. In the paraphrase and in question answering it *consistently maintains this assumption*.

Unfortunately, DISCERN maintains the assumption even when the question supplies the missing information:

```

[ parsing question: ]
DID MARY TRAVEL A BIG DISTANCE TO DOWNTOWN ?
|Mary traveled - big distance *downtown(beach)*|

[ into cue: ]
|$travel $bus Mary *bus-termin(bus-stop)* *downtown(beach)* small _|

[ retrieving from episodic memory: ]
[ map (1,1), (0,0): →(7,1):0.689 →(5,0):0.868 →(5,0):0.971
                    →(5,0):0.967 →(5,0):0.984 →(5,0):0.977
                    →(5,0):0.997 →(5,0):0.981 →(5,0):0.990 ]
[ image units (1,1), (0,0), (5,0): ]
|$travel $bus Mary bus-stop beach small _|

[ generating answer: ]
|Mary traveled - small distance _|
MARY TRAVELED A SMALL DISTANCE .

```

The question assumes that the destination was **downtown**. As the retrieved representation indicates, DISCERN still maintains its initial guess destination=**beach** (although this does not show up in the actual answer). DISCERN treats the new information provided by the question simply as a mistake, and corrects it. Of course, DISCERN could be right. However, in situations like this it would be more plausible to incorporate the new information into the memory trace [Lehnert, 1978; Dyer, 1983]. If the question assumes a fact that was originally left unmentioned, it is quite plausible to incorporate the fact into the memory. Currently DISCERN has no mechanism for doing this. After the trace has been created, it cannot be modified. All discrepancies between the question and the trace are treated as errors in the question.

10.4.2 Role binding errors

Similar stubborn but consistent behaviour occurs when DISCERN generates an incorrect role binding due to a processing error:

```

[ parsing input story: ]
JOHN WENT TO RALPH'S .
|John went - - - Ralph's|
JOHN COMPARED VEGETABLES PRICES .
|John compared - vegetables prices _|
JOHN TOOK SEVERAL VEGETABLES .
|John took - several vegetables _|

```

```

[ into internal rep: ]
|$shopping $grocery John vegetables Ralph's _ _|

[ storing into episodic memory: ]
[ image units (1,0), (0,1), (6,0): ]
|$shopping $grocery John vegetables *Safeway(Ralph's)* _ _|

[ generating paraphrase: ]
|John went _ _ _ *Safeway(Ralph's)*|
JOHN WENT TO *SAFEWAY(RALPH'S)* .
|John took _ big cart _|
JOHN TOOK A BIG CART .
|John compared _ vegetables prices _|
JOHN COMPARED VEGETABLES PRICES .
|John took _ several vegetables _|
JOHN TOOK SEVERAL VEGETABLES .
|John waited _ big _ line|
JOHN WAITED IN A BIG LINE .
|John paid cashier _ _ _|
JOHN PAID THE CASHIER .
|John left _ _ _ *Safeway(Ralph's)*|
JOHN LEFT *SAFEWAY(RALPH'S)* .

[ parsing question: ]
WHERE DID JOHN BUY VEGETABLES ?
|John bought _ _ vegetables where|

[ into cue: ]
|$shopping $grocery John vegetables Ralph's _ _|

[ retrieving from episodic memory: ]
[ map (1,0), (0,1): →(5,5):0.670 →(6,0):0.956 →(6,0):0.992
                    →(6,0):0.992 →(6,0):0.995 →(6,0):0.994
                    →(6,0):0.996 →(6,0):0.995 →(6,0):0.997 ]
[ image units (1,0), (0,1), (6,0): ]
|$shopping $grocery John vegetables *Safeway(Ralph's)* _ _|

[ generating answer: ]
|John bought _ _ vegetables *Safeway(Ralph's)*|
JOHN BOUGHT VEGETABLES AT *SAFEWAY(RALPH'S)* .

```

In this case DISCERN parses the story correctly into the internal representation, but an error occurs when this representation is stored in the episodic memory. Due to inaccuracies in the representation and the role-binding map, the trace is created around unit with grocery-

store=**Safeway** instead of **Ralph's**. DISCERN cannot clean up this error in later processing, because **Safeway** is a perfectly plausible role binding for grocery-store. Its paraphrasing and question answering will be incorrect, but they will be consistent and plausible.

Most of the errors in DISCERN are of this type. They usually originate at the output of the story parser, as it is trying to complete an input story where a number of events are missing. In the rare event that DISCERN confuses *types*, the error is usually immediately corrected by the next module, as we have seen above. However, when an *instance* is confused with another instance of the same type, the correct instance cannot be recovered later. All paraphrase sentences and answers are generated from the same memory representation, and DISCERN consistently maintains its incorrect but plausible conviction throughout the output generation.

10.4.3 Slips of tongue

Isolated production errors are very rare, but sometimes they do occur:

```
[ parsing input story: ]
MARY WENT TO BURGERKING .
|Mary went - - - BurgerKing|
MARY ASKED THE CASHIER FOR FRIES .
|Mary asked cashier - fries -|
MARY ATE A SMALL FRIES .
|Mary ate - small fries -|

[ into internal rep: ]
|$restaurant $fast Mary fries BurgerKing bad none|

[ storing into episodic memory: ]
[ image units (0,0), (0,1), (3,3): ]
|$restaurant $fast Mary fries BurgerKing bad none|

[ generating paraphrase: ]
|Mary went - - - BurgerKing|
MARY WENT TO BURGERKING .
|Mary asked cashier - fries -|
MARY ASKED THE CASHIER FOR
hamburger(fries)*>*hamburger(fries)*>*HAMBURGER(FRIES)* .
|Mary paid cashier - - -|
MARY PAID THE CASHIER .
|Mary seated - - Mary -|
MARY SEATED MARY .
|Mary ate - small fries -|
MARY ATE A SMALL FRIES .
|Mary tasted - bad fries -|
```

THE FRIES TASTED BAD .
|Mary left _ _ _ BurgerKing|
MARY LEFT BURGERKING .

[parsing question:]
WHAT DID MARY EAT AT BURGERKING ?
|Mary ate _ _ what BurgerKing|

[into cue:]
|\$restaurant \$fast Mary *hamburger(fries)* BurgerKing bad none|

[retrieving from episodic memory:]
[map (0,0), (0,1): →(3,3):0.682 →(3,3):0.949 →(3,3):0.990
→(3,3):0.990 →(3,3):0.995 →(3,3):0.993
→(3,3):0.996 →(3,3):0.995 →(3,3):0.996]

[image units (0,0), (0,1), (3,3):]
|\$restaurant \$fast Mary fries BurgerKing bad none|

[generating answer:]
|Mary ate _ small fries _|
MARY ATE A SMALL *hamburger(fries)*>fries>*HAMBURGER(FRIES)* . . .

[parsing question:]
WHERE where DID did MARY Mary EAT eat FRIES fries ? ?
|Mary ate _ _ fries where|

[into cue:]
|\$restaurant \$fast Mary fries *McDonald's(BurgerKing)* bad none|

[retrieving from episodic memory:]
[map (0,0), (0,1): →(4,1):0.687 →(3,3):0.924 →(3,3):0.984
→(3,3):0.982 →(3,3):0.991 →(3,3):0.988
→(3,3):0.993 →(3,3):0.990 →(3,3):0.994]

[image units (0,0), (0,1), (3,3):]
|\$restaurant \$fast Mary fries BurgerKing bad none|

[generating answer:]
|Mary ate _ _ fries BurgerKing|
MARY ATE FRIES AT BURGERKING .

In one of the sentences in the above paraphrase, the fast-food produced by the sentence generator was closer to hamburger than fries for no apparent reason. A similar "slip of tongue" can also occur in the associative mapping in the lexicon:

[parsing input story:]
JOHN WENT TO BROADWAY .
|John went - - - Broadway|
JOHN LOOKED FOR GOOD SHOES .
|John looked - good shoes -|
JOHN TOOK THE BEST SHOES .
|John took - best shoes -|

[into internal rep:]
|\$shopping \$clothing John shoes Broadway - -|

[storing into episodic memory:]
[image units (1,0), (0,0), (1,5):]
|\$shopping \$clothing John shoes Broadway - -|

[generating paraphrase:]
|John went - - - Broadway|
JOHN WENT TO Broadway>Broadway>*BULLOCK'S(BROADWAY)* .
|John looked - good shoes -|
JOHN LOOKED FOR GOOD SHOES .
|John tried John several shoes -|
JOHN TRIED ON SEVERAL SHOES .
|John took - best shoes -|
JOHN TOOK THE BEST SHOES .
|John paid cashier - - -|
JOHN PAID THE CASHIER .
|John left - - - Broadway|
JOHN LEFT BROADWAY .

One of the generated Bullock's was incorrectly mapped on the lexical symbol BROADWAY. Similar errors can also happen in the lexical → semantic mapping with noisy input representations.

The isolated errors almost always occur between two instances of the same meaning. Noise in the two ID units can fairly easily change the identity, but it is much harder to accidentally change the content. When the flip occurs, the output is **exactly** the incorrect instance, rather than some intermediate or non-recognizable representation. These errors are quite similar to semantic paralexia errors in human deep dyslexia, where words with different output form but similar meaning are confused [Coltheart et al., 1980; Caramazza, 1988].

10.4.4 Forgetting and memory confusions

Memory errors occur when the episodic memory becomes overloaded with similar stories. Let us have DISCERN read two almost identical narratives:

[parsing input story:]

MARY WENT TO BULLOCK'S .

|Mary went - - - Bullock's|

MARY LOOKED FOR GOOD JEANS .

|Mary looked - good jeans -|

MARY TOOK THE BEST JEANS .

|Mary took - best jeans -|

[into internal rep:]

!\$shopping \$clothing Mary jeans Bullock's - -|

[storing into episodic memory:]

[image units (1,0), (0,0), (1,0):]

!\$shopping \$clothing Mary jeans Bullock's - -|

[parsing question:]

WHAT DID MARY BUY AT BULLOCK'S ?

|Mary bought - - what Bullock's|

[into cue:]

!\$shopping \$clothing Mary *shoes(jeans)* Bullock's - -|

[retrieving from episodic memory:]

[map (1,0), (0,0): →(3,7):0.676 →(1,0):0.625 →(1,0):0.879
→(1,0):0.776 →(1,0):0.911 →(1,0):0.807
→(1,0):0.919 →(1,0):0.819 →(1,0):0.923]

[image units (1,0), (0,0), (1,0):]

!\$shopping \$clothing Mary jeans Bullock's - -|

[generating answer:]

|Mary bought - - jeans -|

MARY BOUGHT JEANS .

[parsing input story:]

MARY WENT TO BROADWAY .

|Mary went - - - Broadway|

MARY LOOKED FOR GOOD JEANS .

|Mary looked - good jeans -|

MARY TOOK THE BEST JEANS .

|Mary took - best jeans -|

```

[ into internal rep: ]
|$shopping $clothing Mary jeans Broadway - -|

[ storing into episodic memory: ]
[ image units (1,0), (0,0), (3,0): ]
|$shopping $clothing Mary jeans Broadway - -|

[ parsing question: ]
WHAT DID MARY BUY AT BULLOCK'S ?
|Mary bought - - what Bullock's|

[ into cue: ]
|$shopping $clothing Mary *shoes(jeans)* Bullock's - -|

[ retrieving from episodic memory: ]
[ map (1,0), (0,0): →(3,7):0.676 →(1,0):0.858 →(3,0):0.957
                    →(3,0):0.942 →(3,0):0.973 →(3,0):0.956
                    →(3,0):0.977 →(3,0):0.962 →(3,0):0.979 ]
[ image units (1,0), (0,0), (3,0): ]
|shopping $clothing Mary jeans *Broadway(Bullock's)* - -|

[ generating answer: ]
|Mary bought - - jeans -|
MARY BOUGHT JEANS .

```

In the first story, Mary bought the jeans from Bullock's, and in the later story, from Broadway. These stories are mapped on very close on the clothing-shopping map, and the later trace completely wipes out the earlier trace. DISCERN does not remember Mary's visit to Bullock's anymore. When asked about it, DISCERN assumes that Bullock's was mentioned in the question by accident, and answers the question with clothing-store=Broadway. The memory has become overloaded with just two stories, when the stories are sufficiently similar.

Another interesting type of memory confusion occurs when several similar narratives are stored on the same area of a role-binding map, but far enough apart so that the earlier traces are not completely erased. There is a lot of excitatory lateral support on that area of the map, and the retrieval activity may not converge to any one of the stored traces. Instead, the activity peak may develop around some intermediate unit. In this case, a representation is retrieved which is a combination of the stored traces. In other words, DISCERN remembers several distinct features of similar stories, but cannot keep the stories separate.

Chapter 11

Discussion

11.1 DISCERN as a physical model

The DISCERN architecture was not developed with biological plausibility as a major goal. However, much of the architecture is neurally inspired: distributed representations, recurrent networks, ordered pathways between processing modules and feature map layout of information all have correspondence in the physical structures in the brain. DISCERN is an artificial intelligence model, and artificial neural networks are seen as a class of statistical machines, which have several useful properties for natural language processing. A major motivation for our work is to give a better account for high-level phenomena, based on these properties.

It is instructive to analyze where the model is not biologically plausible and why. The overall structure in the performance phase seems as reasonable as it can be, given our state of knowledge about high-level processes. There are separate input/output channels, which share the same semantic component of the lexicon (section 9.5). The tasks of understanding and generating language are performed separately (by the sentences modules), and the different higher-level processes (paraphrasing and question answering) use these modules. Information is processed in recurrent networks sequentially in real time, and it flows through pathways which are organized according to the structure in the data. However, the recurrence is used only for sequencing, not for settling. The FGREP networks compute straightforward transformations. They cannot model reaction times and settling into solutions.

The information is represented distributively, which makes processing robust against noisy input, internal noise, and minor and diffuse damage. With extensive lesions it would also be possible to cause selective impairment on one of the processing functions. Both robustness and selective impairment have been observed in human patients. On the other hand, the syntax of the language is inseparable from the semantics (see section 11.2). It is not possible to lesion the system so that it loses one while preserving the other. However, syntactic and semantic processing appear to be selectively impaired in certain cases of human aphasia [Sarno, 1981].

The greatest strength of the modular FGREP approach as an AI technique is also its most serious problem as a neural model. While the system is fairly plausible in the performance configuration, it is hardly justified to separate performance from training and completely cut the connections between modules during training. This separation technique allows for an efficient, parallel training of the modules, which is a good AI technique. There is no way to avoid cutting connections, as long as we are using backpropagation training.

DISCERN is still a plausible learning model, but at a higher level of abstraction. We are primarily interested in building a neural network model of performance, and we need a

mechanism to set the weights appropriately. Hand coding the weights is far too complex, but fortunately backpropagation to find them for us, if we can come up with a good sample of performance examples. The weights are learned from statistical correlations in the examples in a process that models environmental feedback (training input). At the level of biological neural networks, the mechanism for extracting the correlations and for providing the feedback is not plausible. However, the approach as a whole is plausible at the abstract level. Backpropagation is strong and to date the most utilized method for learning the correlations. Undoubtedly it will be later replaced by biologically more plausible algorithms.

The memory structures are the most plausible components in DISCERN. They do not suffer from the training/performance dichotomy since the self-organization takes place in an unsupervised manner, without a teacher. Their operation directly depends on the physical organization of the hardware. Different memories are laid on different parts of the hardware, and information is presented with value-unit encoding [Ballard, 1986]. Several such maps are known to exist in the central nervous system, e.g. retinotopic maps, tonotopic maps, and also tactile and motor maps [Kohonen, 1984]. It is quite possible that also higher-level information is represented in a similar manner.

The lexicon module in DISCERN can be locally lesioned, and it displays deficits similar to human patients. This suggests that the model successfully represents some of the physical structure underlying the lexical system in the brain. The architecture is based on word maps, where different units are selectively sensitive to different words in the data. Recently it has been found that neurons in the hippocampus respond selectively to visually presented words [Heit et al., 1989]. These response characteristics could be explained by a map-like structure.

The feature map memories in DISCERN still finesse a lot of fine neural structure, and the correspondence to the neuron level is nontrivial. The units and connections in the model do not necessarily correspond one-to-one to neurons and synapses, but rather, to connected groups of neurons. For example, the weight vectors in the lexical and semantic maps of the lexicon and in the episodic memory are used both for input and output, which is not a plausible model of the synaptic efficacies. Similarly, in the hierarchical feature map architecture the units have to determine which lines to pass on to the submap by collecting statistics on the activities on each line. They have to be able to selectively pass on some lines while suppressing others. Obviously this cannot be done by single neurons. Given our current knowledge of the neural processes, there is plenty of room to speculate how the above mechanisms could be implemented, e.g. as connected groups of neurons [Miikkulainen, 1990a].

11.2 Psychological implications

More than biological plausibility, DISCERN aims at psychologically plausible high-level behaviour. It is important that the errors in role binding, memory retrieval and lexical access are plausible in terms of human performance. Several cases were presented in chapter 10, and DISCERN seems to do fairly well in this respect.

On the other hand, the model also makes several predictions about human knowledge organization and the mechanisms underlying knowledge processing. Perhaps the most im-

portant one is the idea of laying knowledge out spatially on maps. We have shown how high-level phenomena such as recency preference of similar memory traces and persistence of unique traces can be explained by the spatial layout of the memory.

Parsing and generating language are performed in separate modules in DISCERN. To some extent these tasks are dissociated in humans as well. It is possible to develop an impairment of one of the functions while the other one is preserved. Understanding language is usually an easier task to learn than generating it. Our model predicts that question answering could be dissociated as well.

However, the sharing of knowledge is inadequate in the model. If DISCERN learns to parse a new type of story, e.g. visit to a dentist, it would still have no idea how to generate a paraphrase of it. The learning in parsing does not carry over to paraphrasing. Knowledge sharing is a general problem in modular systems [Minsky, 1985].

The lexicon implements knowledge sharing to some extent at the very low level. The system could learn a new word in a parsing context, and because of the similarity of this word to other words, could use this for generation as well. High-level knowledge sharing seems impossible without higher level structure, though. If, e.g. question answering would be abstracted from the patterns, and would implement higher-level rules instead, questions about novel representations could be answered. DISCERN, and current PDP systems in general, cannot do this.

There is no explicit representation of syntax in DISCERN. As a matter of fact, syntax is seen as indistinguishable from semantics. Both are learned simultaneously from examples, and coded into the word representations and into the weights of the processing modules. This encoding includes even syntactic markers such as period and question mark. The approach drastically differs from the more traditional views in linguistics, where syntax is often studied as a separate function. Also, data from aphasic patients [Sarno, 1981] and persons grown in isolation [Curtiss, 1988] shows at least a degree of dissociation of these tasks. Unfortunately, DISCERN cannot model this dissociation. However, it does give a reasonable answer to the bootstrapping problem, i.e. how both the syntax and semantics can be learned simultaneously. The processes that learn them do not represent them differently.

11.3 DISCERN as a developmental model

It is fairly difficult to characterize the course of development in a composite high-level task such as script processing, especially when the task is divided into modules which are trained separately. DISCERN can model development at two levels: (1) at the system level, DISCERN predicts restrictions in the order in which the different subtasks must be learned, and (2) at the level of individual modules, the course of learning resembles human learning in limited cases.

DISCERN models a naive language learner who is learning to make sense out of its sensory inputs, rather than one being taught by high-level instruction. The lexical map can be organized before anything else in the system. The map of the possible symbols of the language can be developed just by observing the environment. In other words, it is possible to

recognize the language before being able to understand and use it.

The syntax and the semantics of the language are inseparable, and competence in the language is developed simultaneously with the semantics of its constituents, that is, the task is learned at the same time as the semantic representations are developed.

The episodic memory can be organized during development, but traces which are created early in the process may become inaccessible when the semantic representations change. After the representations settle, the memory organization becomes stable and traces can be reliably recalled.

The different modules of DISCERN independently show interesting developmental stages. The processing modules learn to process the content of words much earlier than they learn to process the IDs. The case-role assignments and the event sequences are semantically correct quite early in the training, but the instances are often confused with each other. The system has trouble learning logical bindings which are not constrained by the semantic properties of the constituents. For example, after the customer orders a particular food, he eats what he ordered. DISCERN might suggest eating something else, but never non-food. Similar problems with logical reasoning have been observed in young children [Osherson, 1974].

The self-organizing process develops the script taxonomy in a way that seems plausible in terms of human learning. The process begins by establishing a gross ordering of the input data, dividing the input into a few large categories [Miikkulainen, 1987], i.e. the most prominent and regular event sequences are recognized as the first rudimentary scripts. For example, the restaurant and the shopping stories are first grouped together, separate from the travel stories. These categories become gradually more refined, as more attention is paid on the details. *The more frequently a certain kind of input occurs in the input data, the more of its details become significant.* The restaurant and shopping stories together are twice as numerous as the travel stories, and what used to be different variations of the shopping-restaurant script eventually become scripts on their own right.

Similar course of development can be seen in organizing the semantic map of the lexicon. Similar word meanings are first grouped together, and the topological order of the groups is established. During this time, there are few distinct meanings in the system, even when the vocabulary is large. The words are overextended. In further training, the representations gradually spread over the whole area of the map, and the fine distinctions of meaning become apparent.

11.4 Making use of modularity

Most of the parallel distributed processing models have relied on capabilities of homogeneous architectures, such as simple backpropagation networks. It seems that in more complex domains, division into subtasks could be a useful approach [Minsky, 1985; Ballard, 1987; Waibel et al., 1988; Ritter, 1989; Jacobs, 1990].

For example, Harris and Elman demonstrate how a simple recurrent network can learn to represent co-occurrence relationships (i.e. role-bindings) in script-based stories [Harris and Elman, 1989]. The stories (various instantiations of the script) are input word by

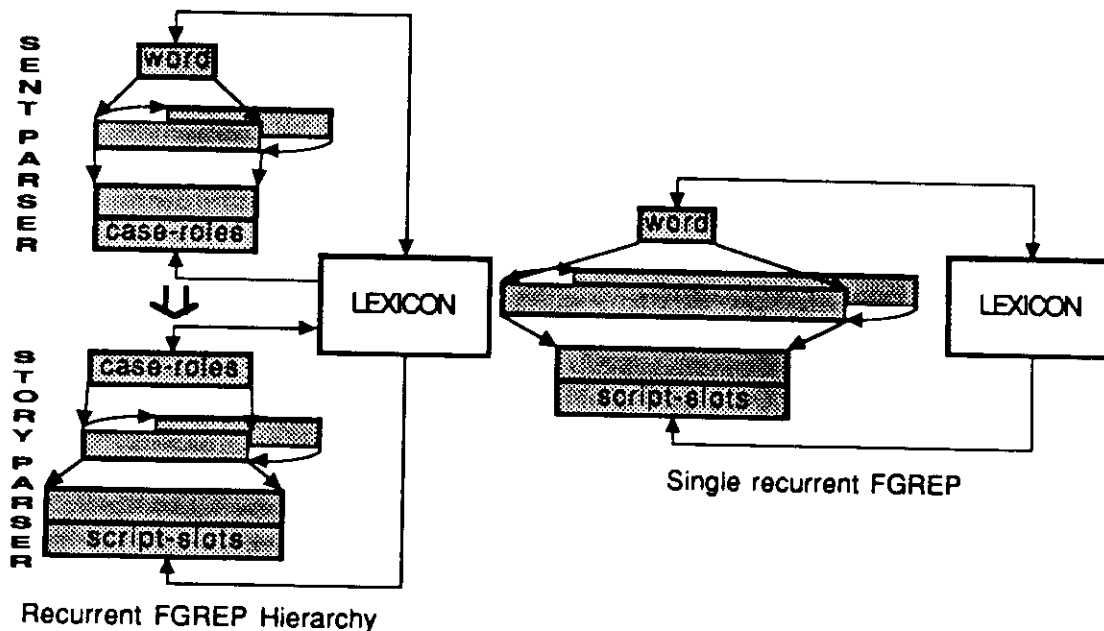


Figure 11.1: **Parsing stories with hierarchical modules vs. a single network.** The DISCERN parsers have a hierarchical interface, with the sentence case-role assignment as an intermediate representation. The single recurrent FGREP network reads a sequence of words directly into a story representation.

word, and the network is trained to predict the next word. The non-variable words of the script are predicted very well, but the correct role fillers are predicted only if they were recently mentioned in the story. In DISCERN, the modular structure efficiently cuts down the information that the subnetworks have to represent in their hidden layers, and consequently, remembering the correct fillers is not problematic.

In general, building the system from *hierarchically organized separable modules* provides two major advantages: (1) the task is effectively divided into subgoals, which is an efficient way to reduce complexity [Minsky, 1963; Korf, 1987], and (2) the system is forced to develop meaningful internal representations, which can be used to add more functions to the system.

A simple comparison of hierarchical networks and a flat network was devised to demonstrate the first point. The hierarchical system consisted of the parser networks of DISCERN, with sentence case-role assignment as an intermediate representation. The flat network consisted of a single recurrent FGREP module with 126 hidden units, giving approximately the same number of connection weights (figure 11.1). Both systems were trained to read script-based stories word by word into the internal slot-filler representation. The hierarchical system achieved a satisfactory level of error about twice as fast as the simple system (table 11.1). The simple network never reached the same level of correct output words, although it was trained five times longer than the hierarchical one.

To be most effective, the division into submodules should take place in a natural way, based on the properties of the task. This makes it *possible to change the function of the system in a modular fashion*. If the intermediate representation is meaningful, it is likely

System	Time	Words	$E_i < 0.15$	E_{avg}
DISCERN parsers	1.5	96	98	.022
Single Rec. FGREP	3.5	91	98	.021
Single Rec. FGREP	7.5	93	99	.013

Table 11.1: **Reducing complexity with hierarchical modules.** The training data consisted of restaurant script based stories, without IDs. Training time is shown in days (wallclock time) on an HP 9000/350 workstation. Learning rate was 0.1 at first, and 0.05 after 3.5 days. The figures for DISCERN parsers indicate the performance of the sentence parser – story parser chain.

System	Time	Words	$E_i < 0.15$	E_{avg}
DISCERN parsers	1.5	96	98	.022
Without FGREP	1.5	96	99	.022

Table 11.2: **Effect of FGREP on training time.** The “Without FGREP” -system uses the final representations of the DISCERN parsers, without modifying them. The same training data and parameters were used as in table 11.1. Training time is given in days on an HP 9000/350 workstation. The figures indicate the performance of the sentence parser – story parser chain.

that additional modules can be added to the system with little modification, using the intermediate level as input or output.

In reading and generating stories, the sentence level provides a natural subgoal. The story networks of DISCERN contain the general semantic and scriptal knowledge which is needed for inferencing, while the sentence networks form the specific language interface. After a story is read into the internal representation, the only information that is actually stored are the role bindings. The knowledge about the events of the script is in the weights of the sentence and event generator networks. Different networks can be trained to answer questions and paraphrase the story from the same role bindings in a different style or detail, *even in a different language* (section 13.1.4).

11.5 The role of the central lexicon

Developing the semantic input/output representations for the FGREP modules in a central lexicon does not seem to incur an extra cost on the training time. The lexicon of the hierarchical script parser system (section 11.4) was fixed at its final state, and the system was trained again from random initial weights without FGREP. Learning was faster at first, apparently because meaningful representations made the task easier. But the fine tuning of performance took longer, because the system could not modify the representations to its advantage. The overall learning time turned out to be about the same as with FGREP (table 11.2).

Adding more modules to the system, all developing the same semantic representations,

System	Time	Words	$E_i < 0.15$	E_{avg}
Parsers / 2 modules	1.0	96	97	.025
Parsers / 4 modules	0.7	96	99	.014

Table 11.3: **Effect of more modules on training time.** The same training data and parameters were used as in table 11.1, but each network was trained on separate workstations in parallel. Time indicates days of parallel training. The figures indicate the performance of the sentence parser – story parser chain.

does not seem to make learning any harder either. The learning in two different systems was compared: (1) the parser and generator modules of DISCERN were trained on four workstations in parallel (top and bottom of figure 5.5), and (2) the two parser modules of DISCERN were trained on two workstations in parallel (the top left and bottom left networks of figure 5.5). The parsing part of the first system reached a satisfactory level of performance 30% *faster* in wallclock time than the mere parsing system, despite the fact that in this case there were two extra modules modifying the representations also (table 11.3). This is surprising because with more modules, the representations have to encode more processing requirements, which at the outset could slow down the convergence. Apparently, *in a system with more modules, the representations become descriptive faster, which speeds up the total learning.*

Currently, the double feature map model of the lexicon is separate from the FGREP lexicon (see section 13.3.2 for combining them). The double feature map model translates between the lexical and semantic representations at the input and output of DISCERN during performance. The FGREP lexicon is maintained by the external training supervisor while the semantic representations are being developed by FGREP.

The training lexicon consists of conceptual word representations. This lexicon could play the role of a more general symbol table, containing representations for hierarchically more complex structures as well. This is essential in modeling formation of new concepts. A reduced description (see e.g [Pollack, 1988]) could be formed for a complex structure, and placed in the lexicon. A reference to the structure could be made using this lexicon entry, and communicated between modules like a word. A first step into this direction has already been taken in the DISCERN system. The internal representation for a story contains script and track slots, which are filled with distributed representations of the different script and track types. These representations stand for higher-order structures (coding information about event order and specific roles), although they are processed like words by the system.

The fact that FGREP can code both meanings of, e.g., *bat* into a single representation is a good demonstration of the power of the mechanism. However, the distinction between the lexical word *bat* and the two concepts associated with it, *baseball-bat* and *live-bat* becomes blurred in this case. It would make more sense to develop separate representations for the distinct meanings of homonymous words.

A modified version of FGREP has been developed to do this [Fellenz, 1989]. When an ambiguous lexical word is read in, an *average* of the alternative concept representations is

input to the network. The network is trained to produce the representation of the correct meaning, e.g. **baseball-bat**, as its output. During backpropagation, only the representation for the correct meaning is modified. The modified FGREP network develops representations for both **baseball-bat** and **live-bat**, and learns to disambiguate between them.

However, in DISCERN the distinction between lexical and conceptual representations is made even more clear in the performance lexicon. The actual input to the system consists of lexical words. The lexicon translates each lexical word into its corresponding concept representation, and the system internally processes only concept representations.

11.6 Robustness and stability

The connectionist modules are never perfectly accurate in their subtasks. A connectionist modular system can only operate if it is stable, i.e. small deviations from the familiar flow of information are automatically corrected. If this is not the case, the noise will accumulate in the system and eventually makes performance impossible.

DISCERN filters out noise in several ways (see section 10.4). Inaccuracies in the output of one module are cleaned up by the module that uses the output, based on the redundancies in the distributed representations. The memory modules clean up by categorical perception. The noisy input is recognized as a representative of a familiar class and replaced by the correct representation of that class. In other words, DISCERN has automatic built-in safeguards against minor inaccuracies and noise. Small deviations do not throw the system off course, but rather, the system filters out the errors and returns to the normal course of processing. In this sense the system is stable.

There are certain errors which the safeguards cannot deal with. If an ID pattern is confused with another ID pattern, in most cases the system cannot recover (section 10.4). Because the IDs are nonredundant, the low-level processes do not have enough information to tell a correct pattern from an incorrect one. Only a high-level monitor process could do it. This process would constantly monitor and evaluate the performance of the low-level networks, and intervene with the execution when the low-level systems get lost.

Currently, DISCERN does not have high-level control at all. DISCERN demonstrates how far it is possible to get in modeling high-level behaviour by combining simple low-level processes. The high-level behaviour in DISCERN emerges from a series of reflex responses. DISCERN is not "conscious" of what it is doing, i.e. it does not have representations concerning the nature of its own representations and processes.

11.7 Question answering

DISCERN was trained to answer simple questions about role bindings in the stories. An interesting issue is, can it generalize question types to new stories? To test this, each track was trained with 2 out of the 3 wh-questions (who, what, where), and tested on the remaining question. Different questions were included for each track of the same script, so that the system had seen all question types within the same script, but not in the context

of the same track. For example, fancy-restaurant stories were trained with who and what questions, coffee-shop with what and where, and fast food with who and where.

Generalization did occur to some extent. The accuracy of the story parser was 85% on the unfamiliar question, while it was 95% on the familiar ones. This seems like a promising result. Generalizing from two cases into third is not easy for backpropagation networks in general. To break the association to special cases it would be necessary to see more variation. If we had more examples to show to the network, very likely it would generalize better.

A particular question type that might generalize well is yes/no-question. This simply requires verification of an event, or a correction of it. For example, asking Did John eat a lobster at MaMaison would be answered John ate a lobster at MaMaison or John ate a steak at MaMaison, depending on the food-slot in the story representation. The task consists of comparing the case-role assignment to the fillers of the story representation, and producing a possibly modified case-role assignment as output. An interesting experiment would be to train the network with different verification situations and see how well it can verify/correct new situations. It is possible to come up with enough examples so that the system might be able to dissociate the question from the context.

However, there are obvious limits to the generalization that is possible. Much of it depends on the structure of the internal representation. To generalize a who-question to another representation, the same role at least needs to be represented at the same assembly. This should be possible to arrange. For the generalization to be meaningful, the two contexts need to have similar structure, and their representations need to be similar, even if the role names are somewhat different. For example, the customer in a restaurant corresponds to the passenger in the train etc.

A more serious limitation is the generalization into novel situations in general, discussed in section 11.8. Unless trained with excessive examples (which might be possible in the case of the verification question), the network has no reason to develop an abstract representation for the question. It only learns to process questions according to statistical correlations in the input. If DISCERN's answer producer module is trained with who-questions about restaurant stories only, and then asked Who bought a CD-player at Radio-Shack in the novel context of a shopping story, the network would simply produce garbage. For generalization at this level, a higher-level, abstract representation for the question is necessary, and at this point it is unclear how this could be learned from the examples.

11.8 Exceptions and novel situations

St. John has studied the limitations of the PDP approach in processing unusual and novel situations in his story gestalt model [St. John, 1990]. Although his architecture is quite different (the story gestalt model is an application of the sentence processing architecture [St. John and McClelland, in press] to sequences of propositions), his findings are very similar to what we have come across in DISCERN. Processing knowledge in these models is based on statistical regularities, and the models cannot handle deviations from the regularities very well. For example, if DISCERN reads a fancy-restaurant story where the food is bad, it will override the input and paraphrase the story with taste=good. This is because in all training

examples, the food in a fancy-restaurant was good. The network learns this fact as part of the track, not as a role binding.

However, if the correlation between fancy-restaurant and good food is broken with counterexamples in the training, the system has to make the taste of food a variable. This way it is possible to train the network to handle deviations to a limited extent [St. John, 1990]. For example, if in 90% of the fancy-restaurant stories the food were good, and in 10% it were bad, the network would develop a strong expectation for good food. After reading a sentence specifying that the food was actually bad, it would change the binding into bad. The overall behaviour in this case resembles garden-path processing, but there is a significant difference. The network does not commit itself to any particular interpretation and then backtrack. All possibilities are kept active at all times, only their relative strengths change.

Training a network to handle exceptional inputs is very expensive. For a long time in the training, the overall error goes down faster by simply ignoring the exceptional cases. Eventually the network has to pay more attention to them and it learns to process them correctly. The less frequent the exception, the harder it is to learn.

PDP systems generalize fairly well into novel situations which are in line with their training. They are good at interpolating within the training data, but they cannot extrapolate outside it very much. For example, having seen **The man ate the lobster** the network might be able to process **The dog ate the lobster**, if both man and dog shared animate qualities in the data. A truly novel role binding, such as **The train ate the lobster**, which goes against all regularities in the training data, would be beyond its capability. The network has no reason to abstract the idea of a symbolic all-or-none role-binding. The bindings only emerge from the statistical correlations of the constituents [St. John, 1990]. As a result, PDP systems can naturally model semantic illusions, where the actual content of the text is overridden by semantically more likely content. But they cannot process truly novel role bindings according to a symbolic higher-level rule. The two approaches are incompatible, as has been pointed out by [van Gelder, 1989].

Chapter 12

Comparison to previous work

12.1 Symbolic models of natural language processing

Several symbolic models have been built based on script theory. The first and most comprehensive is the *SAM* (Script Applier Mechanism) system [Cullingford, 1978], which was developed simultaneously with the theory. *SAM* has several script representations in its memory. It forms a conceptual dependency representation [Schank and Abelson, 1977; Schank and Riesbeck, 1981] of each input sentence and compares this by symbolic pattern matching to the scripts that it knows. If this representation constitutes part of a script, the representation of that script is instantiated in the memory, and rest of the inputs are matched against this script. If there are events in the script between two stated events which must have occurred, they are simply inferred to have happened and included in the story representation.

If new inputs cannot be matched with the script representation, the program tries to explain the input as a deviation. An example of this is e.g. failure to get service in a restaurant. The system can also deal with multiple active scripts at the same time, with some nontrivial interactions. For example, having one's pocket picked in the train prevents paying in the restaurant later that evening. *SAM* generates fully expanded paraphrases in a couple of different languages, as well as short summaries of the stories, and supported simple question answering.

FRUMP (Fast Reading Understanding and Memory Program) [DeJong, 1979] is another script processing program with slightly different objectives. *FRUMP* was designed to skim newspaper stories about stereotypical occasions like vehicle accidents. It uses a "sketchy script" representation, which consists of the most important events and roles only. The sketch has a number of slots which *FRUMP* tries to fill while reading the story. For example, *FRUMP* might be looking for the location of accident, type of vehicle, number of people killed and injured etc. This top-down approach allows the program to read very fast: irrelevant inputs are simply ignored. *FRUMP* can also later read an update of the original article, and fill in slots which were unspecified in the first article. The program demonstrates its understanding by generating a summary of the sketchy script with its slots filled with the information from the articles.

These two symbolic programs clearly demonstrate the power of script theory in processing texts about everyday events. The models are capable of quite impressive behaviour in their domains. *DISCERN* does not attempt quite as extensive modeling of script processing. The system is trained with several different kinds of scripts, and it has to be able to choose the one relevant to the current text. But it is assumed that the input stories are always based on single scripts, with no deviations or script interactions in the same sense as above. It should

be possible to train the system to deal with cases like no service in a restaurant, or robbery during travel. The system can learn to process regular sequences of events, and deviations and interactions could be represented this way.

The main goal of the experiments was to demonstrate that modular FGREP networks are a plausible approach for modeling high-level cognitive tasks such as story paraphrasing and script recognition. The main advantage of this approach is that processing is learned from examples. The architecture is not committed to any particular data or knowledge structure; the same system can learn to process a wide variety of inputs, depending on the training data. A network which was trained to paraphrase restaurant scripts can learn to paraphrase travel stories, shopping stories etc. just as well.

This contrasts with the traditional symbolic artificial intelligence models, where the processing instructions must be hand-crafted with particular data in mind. These symbolic models cannot learn from statistical properties of the data, they can only do what their creator explicitly programmed them to do.

For example, symbolic expectations must be implemented as specific rules for specific situations [Dyer, 1983; Schank and Abelson, 1977]. Generalization into previously unencountered inputs is possible only if there exists a specific rule that specifies how this is done. Representations of these rules and their applicability is often very complex. This explicit rule-based approach seems unnatural, given how immediate and low level such operations as expectation and generalization are for people.

In the neural network approach, expectations and generalizations emerge automatically from the distributed character of processing. Knowledge for this is *based on statistical properties of the training examples, extracted automatically during training*.

On the other hand, symbolic models currently go far beyond simple script instantiations in modeling story understanding. Complex texts contain references to several scripts and multiple scripts can be active at the same time. Scripts are used to connect the pieces of the text together, i.e. they are tools in building a complete representation of the story. However, stories strictly based on straightforward script instantiations are not particularly interesting. Any reasonable story contains unusual events or events which deviate from the script, and that is often what makes the story worth telling. Symbolic systems have been developed which deal with these issues [Schank and Abelson, 1977; Dyer, 1983; Alvarado et al., in press], based on higher-level knowledge structures and processes, such as goals, plans, affects, beliefs, argument structures, etc.

Current PDP systems do not yet attempt story understanding at this level. They are very good at dealing with regularities, but do not easily lend themselves to processing the unusual or the unexpected (see section 11.8). As long as the required inferences are based on statistical regularities they can be well modeled with PDP. It is quite possible to build connectionist systems which process stories with multiple scripts (see e.g. [Sharkey et al., 1986]), or even several simultaneously active scripts. Deviations from the ordinary events are harder to implement. This would require a higher-level monitoring process, which recognizes deviations and builds separate representations for them, possibly using techniques like plan analysis [Lee, 1990; Dolan, 1989]. However, planning requires "dynamic inferencing" [Touretzky, 1989a], i.e. putting several pieces of information together to form genuinely

novel information, not just pattern transformation. Dynamic inferencing is currently an open problem in PDP research.

12.2 Parallel distributed models of NLP

Very few PDP models of natural language understanding have been built so far. Most of the artificial neural network research has concentrated on studying the mechanisms of networks, without a very clear goal or motivation at a higher level. Isolated, low-level cognitive tasks have often been used as a domain, but emphasis has been more on demonstrating specific techniques than on modeling the task itself. Some of the few models which concentrate on natural language are discussed below.

12.2.1 Sentence processing models

Most work has been done on sentence processing models. McClelland and Kawamoto's model for *assigning case roles to sentence constituents* [McClelland and Kawamoto, 1986] is perhaps the best known, and also served as inspiration for the sentence processing model in this research. McClelland and Kawamoto's architecture was quite different from the one presented in this thesis. The word representations were hand coded as a collection of microfeatures (see section 12.6.1) and remained fixed throughout the experiments. The network consisted of two layers of binary-valued stochastic units. The input layer was divided into assemblies representing the syntactic constituents of the sentence, and the output layer into assemblies representing the case roles. The words were represented in the assemblies as outer products with themselves (in the input) or with their head word (in the output). The connection weights were adjusted according to the perceptron convergence procedure.

The primary goal of McClelland and Kawamoto's experiment was to teach the system to assign correct semantic roles to the words, based on their syntactic role and the semantic context within the sentence. Our sentence parser network attempts to do the same, but from sequential word-by-word input, while simultaneously developing the word representations in an external lexicon. McClelland and Kawamoto's system also performed semantic enrichment of the representations at the output, as well as disambiguation of the different senses of *bat* and *chicken*, which had different representations in their data. These tasks are not done in the sentence processing module in DISCERN, but they can be performed by the performance lexicon in the lexical-semantic mapping.

St. John and McClelland's sentence processing model [St. John and McClelland, 1989; St. John and McClelland, in press] aims at explaining how syntactic, semantic and thematic constraints are combined in sentence comprehension, and how this knowledge can be coded into the network. The model is based on microfeature encoding of the word representations. The first half of their system is a sequential input backpropagation network similar to [Jordan, 1986]. This network reads the phrases of the sentence one at a time, producing a "sentence gestalt" at its output (figure 12.1). The gestalt is basically a hidden layer representation of the whole sentence. The second half of the system (a basic three layer backpropagation network) is trained to answer questions about the sentence gestalt.

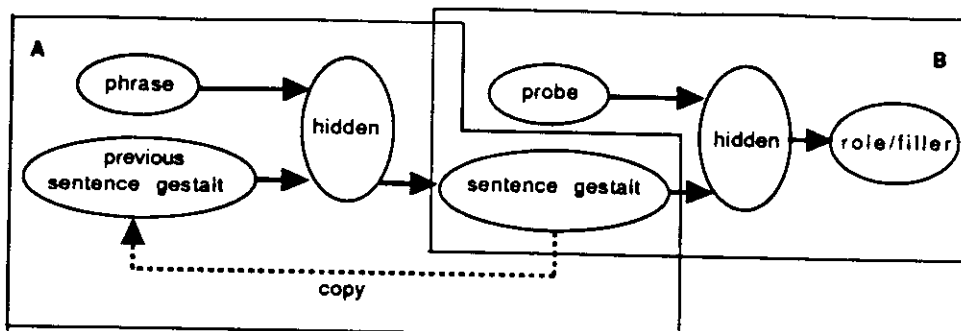


Figure 12.1: The basic architecture of the sentence processing network of [St. John and McClelland, 1989]. The A subnetwork reads the phrases of the sentence sequentially into the sentence gestalt representation. The B subnetwork answers questions about the event represented by the gestalt. Printed with permission.

It is possible to query the gestalt for roles which were unspecified or underspecified in the sentence, even before the reading is complete. The network produces the most likely answer, based on stereotypical knowledge it has extracted from training examples. For example, after reading *The busdriver ate the food*, the first network has produced a sentence gestalt. The gestalt is input to the second network together with the representation of a query, e.g. *patient?*. The second network produces the representation of the answer, *patient=steak* as its output. Because busdrivers usually eat steaks in the data, the system produces this as the answer.

St. John and McClelland's model shows how useful thematic knowledge of this kind can be injected into the system, by training it with queries and by hand-coding the micro-feature representations. The model can answer queries which require information beyond what is strictly provided by the sentence. However, the system cannot be easily used as modular building block in more complex processing. The network's knowledge is *implicit* and *noncompositional*, and can be brought out only by querying the gestalt in the system. To support modularity, the knowledge structures would need to be explicit in the output representation. In FGREP, we are interested in sentence processing as a subtask in language understanding, and we are concerned with integrating sentence processing into the larger context. The sentence case-role representation is publicly interpretable and can be used as input to other modules.

12.2.2 Script processing models

A number of connectionist models of script application have also been proposed. In these models, the recognition of the appropriate script, its instantiation and inferences automatically emerge from the input story representation in a distributed fashion. These models are not usually aimed to be script processing systems. They typically take some part of the task and apply connectionist mechanisms to it. Connectionist mechanisms for processing causal sequences [Golden, 1986], connecting multiple scripts [Sharkey et al., 1986], role binding [Dolan and Dyer, 1987], and learning the script taxonomy from examples [Miikkulainen, in

press] have been proposed.

Harris and Elman have demonstrated how a sequential backpropagation network can learn to represent *co-occurrence relationships in script-based stories* [Harris and Elman, 1989]. The stories (various instantiations of the script) are input word by word, and the network is trained to predict the next word. The network learned to predict the non-variable words very well, but the correct fillers only if they were recently mentioned in the story. The authors claim that the bindings took place not between two different occurrences of the variable, but between the variable and a global state. In other words, the hidden layer of the network learned to represent the role bindings throughout the story. The system was not intended as a story processing system, but it nicely demonstrates the principle used also in DISCERN: the script-based story can be represented in terms of its role bindings, constructed incrementally from the input.

St. John has extended the St. John and McClelland sentence processing model (discussed in section 12.2.1) into processing script-based stories [St. John, 1990]. The basic architecture and operation of this "story gestalt" network is very similar to the sentence processing model. Instead of phrases the network now receives a sequence of propositions as its input. The network is trained to build a gestalt of the whole story by requiring it to answer questions. The questions specify the predicate of a proposition (a verb or a description predicate), and the network is required to produce a representation of the whole event, filling in information such as agent, patient, attribute etc. The proposition representation is localist and role-specific, i.e. there is one unit for each concept in each role.

The sentence gestalt model learns scripts in the sense that it can answer questions about propositions that were not specified in the input sequence. Any regularity in the input data is utilized to draw the inferences, such as stereotypical actions by actors, events that occur together in a sequence etc. These regularities are constraints which are brought together to form an interpretation of the text.

The sentence gestalt architecture provides an excellent tool for analyzing the properties and limits of inference in distributed architectures. The main result is these models give a good account of inferences based on regularities, but cannot apply their knowledge to novel or exceptional situations. This result seems applicable to distributed models in general, including DISCERN (see section 11.8). The gestalt model illustrates the process of learning and using schematic knowledge, but it is not an AI model of script-based understanding.

Our goal is to model natural language understanding in the same sense as the symbolic artificial intelligence models do. This requires parsing and generating natural language, building and processing internal representations, and dividing the task into structured sub-tasks. We want to demonstrate understanding of multisentential connected text. In this sense, our model aims at an order of magnitude higher level than previous distributed models.

12.2.3 Scripts in symbolic vs. PDP processing

There is an important distinction between the scripts (or more generally, schemata) in symbolic systems such as [Schank and Abelson, 1977; Cullingford, 1978; Dyer, 1983], and

scripts in PDP such as DISCERN and [St. John, 1990; Lee et al., 1989]. In the symbolic approach, the script is stored in memory as a separate, exact knowledge structure, coded by the knowledge engineer. The script has to be instantiated, i.e. the schema memory is sequentially searched for the appropriate structure that matches the input. After instantiation, the script is in the memory and later inputs are interpreted in terms of this schema.

In PDP, the schemas do not have explicit representations at all. The schemas are not hand coded in, they are automatically extracted from input examples. Any statistical regularity in the example forms a basis for a schema. They are simply statistical regularities, stored in the processing weights of the network. Every input is automatically matched to every regularity *in parallel*, and there is no all-or-none instantiation. The strongest, most probable correlations will dominate, depending how well they match the input, but all of them are simultaneously active at all times.

The correlations imbedded and used in the PDP systems vary in scope from co-occurring fillers of case-roles to long sequences of events, even to regularities in the goal-plan level. Immediate and intuitive processing, such as script-based inference and simple goal/plan inference, is well-suited to the PDP approach. However, when sequential reasoning becomes necessary, PDP systems usually fall into the "Touretzky tarpit". It is possible to implement e.g. basic symbolic operations such as LISP CAR and CDR, or even a production system with connectionist techniques (which is an important result), but such reimplementations are very inefficient and do not lead to much insight [Touretzky, 1986; Touretzky and Hinton, 1986].

12.2.4 Linguistic models

A number of models of learning *syntactic and phonological aspects* of natural language have also been developed. Rumelhart and McClelland reported experiments on learning past tense forms of English verbs [Rumelhart and McClelland, 1987]. Their model, which had no explicit representation for rules, agreed remarkably well with psychological studies of past tense learning. Plunkett and Marchman recently demonstrated similar behaviour with a more general backpropagation-based architecture [Plunkett and Marchman, 1989]. Sejnowski and Rosenberg showed how a basic three-layer backpropagation network could learn to pronounce English words in a human-like manner [Sejnowski and Rosenberg, 1987]. Although the impressive behaviour of their model relied heavily on pre- and postprocessing, the result is a good demonstration of the backpropagation learning process. Touretzky has developed a model for sentence PP attachment [Touretzky, 1989a] and a first implementation of Lakoff's theory of connectionist phonology [Touretzky, 1989b], [Lakoff, 1989]. Lakoff has also proposed cognitive topology as a new linguistic theory with connectionist foundations [Lakoff, 1989]. Some of the basic ideas of cognitive topology have been implemented in [Regier, 1989].

DISCERN research does not deal with syntactic or phonological issues. The input/output is syntactically very simple, and a number of issues have been finessed (possessive forms, tense, relative clauses etc). In principle the system could learn to process more complex syntax, as regularly occurring patterns of words in the examples (see also section 13.1.3).

But syntax does not have an explicit representation in DISCERN.

12.3 Localist models

Localist models, where items are represented by single nodes, share much of the representation philosophy of symbolic systems, and they are fundamentally different from the distributed approach [Derthick and Plaut, 1986; van Gelder, 1989]. At the core of these models there is a semantic network, which is a simpler version of semantic networks in symbolic systems [Quillian, 1967]. The localist models do not have labeled arcs in the networks like semantic networks. Instead, the localist models are characterized by spreading activation, a general method for retrieval, variable binding and inferencing in these networks.

In pure spreading activation, continuous values are spread through weighted links between nodes [Anderson, 1983; Shastri, 1988; Waltz and Pollack, 1985; Cottrell and Small, 1983]. The initial activity is supplied by instantiating some of the nodes, corresponding to e.g. words in the input text. The activity spreads automatically through the weighted links, and eventually settles into a stable pattern. The collection of the most highly activated nodes constitute the interpretation of the input. The propagation mechanism is extremely simple and general, but also weak because it lacks structure. Numerous spurious paths are created and severe crosstalk problems occur through uninteresting connections.

Marker passing is an extension of pure spreading activation, where discrete symbols are propagated over the network instead of simple activation values. Instead of simple weighted links, multiple link types are used to represent different relations [Charniak, 1986; Hendler, 1988; Granger et al., 1986; Fahlman, 1977]. The markers generate paths through the network, representing various constraints posed by the input. Where the paths intersect, the constraints are satisfied, and the intersection nodes are taken as the interpretation of the input. The markers may contain various kinds of information such as the origin of the marker, the type of constraint it stands for, etc. However, to utilize the structure in the markers, complex high-level mechanisms, "path checkers" are needed to evaluate the alternative solutions. Often the complexity of such filter mechanisms outdoes the advantage of the simple inferencing mechanism.

There have been efforts to combine the advantages of pure spreading activation and marker passing. Lange and Dyer [Lange and Dyer, 1989] and Lange [Lange, in press] describe a system for inferencing and role binding which propagates simple markers, called signatures. The signatures are real values which uniquely identify their source. Crosstalk is eliminated by using gated links (allowing only signatures matching prototypical role fillers to propagate through) and preventing propagation from nodes where different signatures converge. The path selection is performed by simultaneous spreading activation process, where evidential activation is propagated through the network along the signatures. The signature path along the strongest evidential activity selects the winning interpretation without a complex path checking process.

Because they are based on relaxing the activity over time, the localist models can give a plausible explanation for semantic priming effects and timing in cognitive tasks. They offer knowledge-level parallelism (i.e. they can do several distinct inference paths in parallel) and

varying commitment to knowledge structures. Spreading activation and marker passing are simple, universal mechanisms, independent from the content of the network they operate on.

On the other hand, they suffer from a number of shortcomings. The network structure and link weights must be coded in by hand, a process which has a strong ad hoc flavor. They do not support learning, and cannot sustain damage very well. If a node is destroyed from the network, all knowledge associated with it is lost. The reasoning mechanism is general, but the network structure needs to be extremely specific, which makes scale-up into large problems difficult.

It is possible to view the localist network structure as a superstructure to the distributed level. The low-level operations and statistical learning is performed in distributed modules, which are connected in a semantic network. Higher-level reasoning and metaoperations take place at the localist network. Such approach is proposed e.g. by [Dyer, 1990] and [Sumida and Dyer, 1989] (see section 12.6.2).

12.4 Models of the lexicon

The lexicon in symbolic NLP systems is a list of word symbols and phrasal patterns, with pointers to conceptual memory. The memory contains syntactic and semantic knowledge about the lexicon entry in the form of declarations, or procedures which specify how the word should be interpreted in different environments [Zernik, 1987; Arens, 1986; Dyer, 1983]. Phrasal patterns can be recognized as easily as individual words. This knowledge has been explicitly programmed into the system with specific examples in mind. The symbolic lexicons are intended to model the *processes* of lexical access, not the physical structures that implement the processes. Consequently, these models lack the capacity to account for lexical errors in human performance, as well as lexical deficits in acquired aphasia.

A number of connectionist models of lexical disambiguation have been proposed [Cottrell and Small, 1983; Waltz and Pollack, 1985; Gasser, 1988; Kawamoto, 1988; Gigley, 1988]. These models aim at explaining lexical processing with low-level mechanisms, and can better account for the timing of the process, as well as for certain types of performance errors and deficits. However, they are still primarily process models, detached from the physical structures. They are designed as controlled demonstrations, not as building blocks in larger NLP systems.

The performance lexicon in DISCERN is a practical AI implementation of a general computational model of the human lexical system. This model aims at plausibility at the level of *physical structures* such as maps and pathways. The model is based on current cognitive neuroscience theories [Caramazza, 1988; Warrington and McCarthy, 1987; Warrington, 1975] and accounts for several documented lexical deficits in acquired aphasia and dyslexia.

On the other hand, the DISCERN lexicon is an integral part of a larger language understanding system. In terms of the symbolic lexicon models, the lexicon contains both the symbol memory and the conceptual memory, and implements a mapping between them. However, the lexicon is based on distributed representations of word symbols and word semantics. The semantics are based on the statistics of possible environments rather than

explicit coding of phrasal patterns. The memory organization and the mapping are formed in an unsupervised self-organizing process, based on examples of co-occurrence of the word and its meaning. The lexicon is seen more like a filter, which transforms an input word symbol into its semantic representation, and vice versa.

12.5 Models of episodic memory

Scripts in symbolic NLP contain specific information about specific sequences of events. Variations in the sequence are coded explicitly as tracks. There is no abstract information about the structure of the event sequence, such as information about the goals of the actors and the various ways to achieve them. Scripts do not share structure at the abstract level.

However, people sometimes confuse events from different scripts, e.g. events from visiting a doctor with those of visiting a dentist [Bower et al., 1979]. Also, an episode sometimes reminds a person of another episode which occurred in a completely different context, but which had similar abstract structure [Schank, 1982]. If episodes are stored in the memory simply as instantiations of known scripts, it is very difficult to explain interactions between similar memory traces. Episodic memory needs to be organized in terms of similarities and generalizations at the abstract level. These structures need to be dynamic, based on the experience [Schank, 1982].

First model to use dynamic memory structures was Lebowitz' IPP (Integrated Partial Parser) [Lebowitz, 1980]. This program reads newspaper stories about international terrorism and organizes its memory in terms of generalizations it makes. Initially, IPP contains a number of precoded schemata, called s-MOPs (simple Memory Organization Packets). These describe events in an abstract level, and they are necessary for understanding the stories. They contain information that is often left out in the text. For example s-EXTORT describes extortion in abstract terms, without specifying the actions or actors.

After a few texts have been read and indexed under the same s-MOP, IPP forms more specialized knowledge structures called spec-MOPs. These are generalizations of the texts based on similarities between them, and they form a network under the s-MOP. For example, s-EXTORT might point to spec-TAKEOVER and spec-KIDNAP, which again might have spec-BUSINESSMAN as a sub-spec-MOP. Spec-KIDNAP was formed by noticing that extortion in many cases involves taking hostages. Spec-BUSINESSMAN was formed because in many cases the hostages were businessmen. IPP also constantly evaluates the validity of the spec-MOPs it has created. Invalid generalizations are abandoned, and the memory structure reorganizes dynamically with experience. The spec-MOPs store information which is common for several texts, and they organize memories of individual stories. Under each spec-MOP, it suffices to store only the details which differentiate between the texts within the spec-MOP.

The memory in IPP is organized in several levels of abstraction. Reminders between similar experiences can take place through the abstractions. A kidnapping story could remind IPP of a takeover story because they are both special cases of s-EXTORT.

The CYRUS model [Kolodner, 1984] concentrates on issues in maintaining organization

in the dynamic episodic memory, and it also presents a theory of reconstructive retrieval from such memory. CYRUS reads descriptions of experiences of the former Secretaries of State Cyrus Vance and Edmund Muskie. The memory is organized in schematic categories called E-MOPs, or episodic memory organization packets. Each E-MOP contains a set of norms, i.e. a list of features which are common for episodes in the category. An E-MOP for diplomatic meetings contains norms such as: participants are foreign diplomats, topics are international contracts, goal is to resolve disputed contract etc. Under each E-MOP there are tree-like structures which index the individual episodes in terms of their differences. For example, participants and topics serve as indices to diplomatic meetings, differentiating between meetings with Gromyko and Begin, and meetings about SALT and Camp David accords.

The initial memory organization for CYRUS is hand-coded containing E-MOPs for diplomatic meetings, briefings, trips, state dinners, speeches and flying. The memory is reorganized dynamically as more episodes are read in. If the episodes contain unique features, new indices are created using these features. Such an index could be e.g. the location of the meeting. When a new episode has the same value for an index as an earlier episode, e.g. two meetings with Begin, a new E-MOP is created. The similarities of the two meetings constitute the norms of the new E-MOP and the differences constitute its indices. If several episodes under the same E-MOP share a common feature, this feature can be incorporated into the norms of the E-MOP. Initial norms can also turn out to contradict the majority of episodes, and they can be removed.

Memory recall in CYRUS is reconstructive. Individual episodes are not stored as large chunks of facts which are returned as complete entities. Instead, retrieval is a process which constructs the full episode from the indices and norms under which the episode is stored. Often this requires traversing the memory structures and searching for the episode using various strategies.

IPP, CYRUS and other schema-based models of memory such as OCCAM [Pazzani, 1988] form an important class of memory models. Starting with an initial set of general schemata, these models dynamically extend, organize and reorganize the memory, based on the similarities in the episodes. Much of the regular information about individual episodes is stored in the structure; only the most unique features need to be explicitly stored. Retrieval then requires traversing the appropriate memory structures and reconstruction.

Although the above models and the episodic memory in DISCERN are based on very different mechanisms, there are intriguing similarities between them. The DISCERN memory is also organized according to similarities in the stories, and the similarities are more abstract than simple fillers. The higher levels in the hierarchy represent abstractions of individual stories. Much of the information is stored at higher levels of the hierarchy, and only the individual differences (role bindings) need to be stored at the lowest level. Retrieval requires traversing the hierarchy and reconstructing the complete representation from information at multiple levels.

The main difference between DISCERN and the symbolic models is that the organization in DISCERN is based on statistical similarities in the data, whereas the symbolic models are organized according to individual examples. IPP, CYRUS and similar models begin with

predetermined initial structure, and the memory structure increases indefinitely as more experience comes in. In contrast, the hierarchical feature maps are organized with a large number of examples before any traces are created. The whole organization is extracted from examples; it is not necessary to supply basic templates or abstract MOPs to bootstrap the system. The result is a stable layout of the whole space of possible experiences. If an input experience does not fit to the existing structure, it cannot be stored correctly. In principle, the organization could adapt to individual episodes, but this would occur very slowly, requiring a number of presentations.

The symbolic models are process models, and they do not give a very good account of performance errors and impairments that people exhibit. In contrast, the episodic memory in DISCERN specifically aims to be a physical memory model, and several memory effects automatically result from the physical layout. Recency effect, preservation of unique traces and forgetting are an integral part of the model. It is also possible to lesion the system, and selectively loose access to particular types of traces. The symbolic models have not addressed these issues.

12.6 Issues in artificial neural network modeling

The artificial neural network approach brings out a whole set of new issues in cognitive modeling. Many of these issues arise from the subsymbolic nature of processing. Role binding, sequential processing, modularity and one-shot storage are trivial in symbolic systems, but their subsymbolic implementation requires elaborate techniques. On the other hand, issues like semantic representation, probabilistic inference and type/token processing are important in symbolic systems also, but they have a very different character in neural network modeling. In this section, the techniques and methodology of DISCERN in dealing with these issues are discussed and compared with related research.

12.6.1 Methods for forming distributed representations

While the advantages of distributed representations are fairly well understood, an important issue is where do the distributed representations for the input/output items come from.

A popular approach for forming distributed representations is *semantic feature encoding*, used e.g. by McClelland and Kawamoto in the case-role assignment task [McClelland and Kawamoto, 1986] (see also [Hinton, 1981]). Each concept is classified along a predetermined set of dimensions such as human-nonhuman, soft-hard, male-female etc. Each feature is assigned a processing unit (or a group of units, e.g. one for each value), and the classification becomes a pattern of activity over an assembly of units (figure 12.2).

This kind of representation is meaningful by itself. It is possible to extract information just by examining the representation, without having to have a trained network to interpret it. Several different systems can directly use the same representations and communicate using them.

On the other hand, such patterns must be pre-encoded and they remain fixed. Perfor-

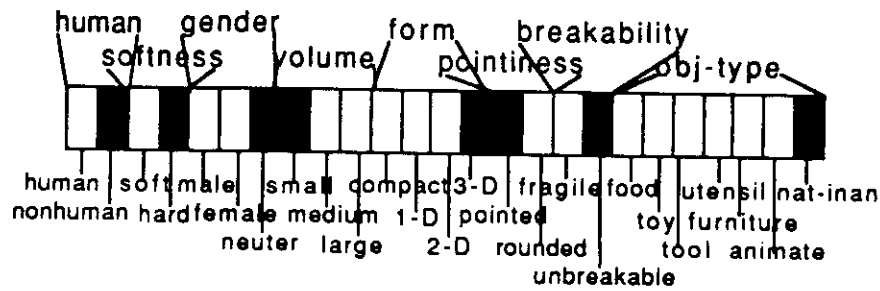


Figure 12.2: **Semantic feature encoding of the concept rock** (after [McClelland and Kawamoto, 1986]). A group of units is assigned to each semantic feature. The units stand for different values (bottom of figure) of the semantic dimensions (top). Black unit indicates that the value is on, white off.

mance cannot be optimized by adapting the representations to actual task and data. Because all concepts must be classified along the same dimensions, the number of dimensions becomes very large, and many of them are irrelevant to the particular concept (e.g. gender of rock). Deciding what dimensions are advantageous to use is a hard problem [van Gelder, 1989]. There is also the epistemological question of whether the process of deciding what dimensions to use is justifiable or not. Hand coded representations are always more or less ad hoc and biased. In some cases it is possible to make the task trivial by a clever encoding of the input representations.

Developing internal representations in hidden layers of a backpropagation network avoids these problems. The network which learns family-tree relationships in [Hinton, 1986] is a good example of this approach. This network consists of input, output and three hidden layers (figure 12.3). The input and output layers are localist, i.e. exactly one unit is dedicated to each item. The hidden layers next to the input and output layers contain considerably fewer units, which forces these layers to form compressed distributed activity patterns for the input/output items. Developing these patterns occurs as an essential part of learning the processing task, and they end up reflecting the regularities of the task [Hinton, 1986].

Another variant of the same approach is presented in [Elman, 1990; Elman, 1989]. A simple recurrent network is trained to predict the next word in the input word sequence. The hidden layer of the network develops structured representations for the words based on the regularities in how the words are used in the sequences.

This approach does not address the issue of *encoding input/output representations*. Hinton's and Elman's systems do not deal with the representations per se; they develop as a side effect of modifying the weights to improve the task performance. The patterns are not available outside the network, and they are not used in communicating with the network. Moreover, since both penultimate layers develop their activity patterns independently, each item has two different representations: one as an input and another one as an output item. These activity patterns are *local, internal processing aids* more than input/output representations which can be used in a larger environment.

Pollack [Pollack, 1989] has proposed a method for forming *compressed representations*

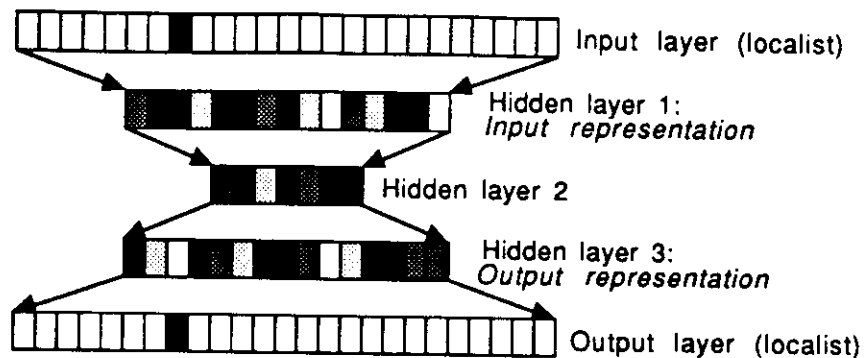


Figure 12.3: **Developing internal representations in hidden layers.** The activity of a single unit in the input layer is propagated through the input weights to the first hidden layer. The resulting pattern (indicated as grey-scale values) constitutes the input representation for this input item. The pattern in the third hidden layer, which results in a single unit being turned on in the output layer, is the output representation for that item. These representations are in general different even if the input and output items are the same.

of recursive structures. The hidden layer pattern of an autoassociative network forms a compressed representation of the input/output layers. A potentially infinite hierarchical data structure, such as a tree, can this way be compressed into a fixed size representation. The tree can be later reconstructed by loading the compressed representations into the hidden layer. The next level of the tree can then be read from the output. This method does not address how the representations for the terminal symbols should be formed.

Lee et al. [Lee et al., 1989] describe a method for developing distributed representations, which are both descriptive in that similar words have similar representations, and autonomous in that they can be used in a number of different tasks. This is accomplished by developing the representations *off-line in a recursive autoassociative network* (similar to Pollack's). The network is trained with triples consisting of representations for a proposition, a case-role, and the word which fills that case role in that proposition. Distributed representations are simultaneously developed for both propositions and words, in two separate networks. The triples become encoded into the representations and they can be read out from the representations with a decoding network. This technique takes an engineering approach to developing representations. They are not learned automatically, but they are manufactured separately in special modules and with special data, designed exclusively for this purpose. Once manufactured, the representations are fixed and cannot adapt to task and data.

In the *FGREP* approach representations for symbols are developed automatically while the network is learning the processing task. Since the representations are adapted according to the backpropagation error signal, they end up coding the properties of the input elements that are most crucial to the task. The representations are global input/output to the system and they are stored in an external network (a lexicon), which guarantees unambiguity and makes communication using these representations possible. The method does not create recursive representations which could be expanded into constituents, but expectations for possible processing contexts can be automatically created.

12.6.2 Processing types and tokens

The ID+content mechanism is a first step towards grounding symbolic reasoning in parallel distributed processing. Any symbolic system needs to be able to deal with two kinds of information: (1) semantics of symbols, i.e. knowledge about the properties of symbols and relationships between them, and (2) identities of symbols, based on some unique surface-level tag, e.g. sensory perception of the referent (see also [Harnad, 1989]). The semantic knowledge is necessary for guiding the processing in a meaningful way, and the identities are necessary for logical reasoning and manipulation.

For example, suppose the system has the semantic knowledge that for some class of objects C , if $A \in C$, $property_1(A) \wedge property_2(A) \Rightarrow property_3(A)$, and it knows the facts $property_1(A_1)$ for $A_1 \in C$ and $property_2(A_2)$ for $A_2 \in C$. Even if this knowledge is statistical (i.e. is not exact but is true with a high probability), to draw the conclusion, the system must be able to verify that in fact $A_1 = A_2$ exactly, not just that they have similar statistical properties.

A common approach to symbolic modeling is to explicitly separate the two kinds of information. The semantic knowledge is maintained in “types”, and each symbol is created as a “token”, an instance of some type. The types form semantic hierarchies, the instances inherit the properties of parent types, and also accumulate specific properties of their own during processing. Whether implemented in a symbolic semantic network [Quillian, 1967] or in a semantic network/PDP hybrid [Sumida and Dyer, 1989], in effect there are two separate systems with a very complex interaction.

The architecture proposed by Sumida and Dyer consists of two levels of representation and processing. At the macro level, the system is a semantic network, representing knowledge about structure and role relations. The nodes of the semantic network are PDP ensembles, which can store multiple instances. The goal of the architecture is to be able to store multiple instances and provide PDP-type generalization between them, while maintaining the parallel inference mechanisms of semantic networks.

In the ID+content approach the identity and semantic content are *kept together in a single representation* and processed through the same pathways and structures. This is essential in building modular systems, because it allows modules to be connected with simple pathways. All the information is included in the output layer, and can be directly used as input in another module. Part of the representation is treated as the logical ID, and the rest is interpreted as the semantic content. The system is trained to process both parts simultaneously.

However, it turns out that processing identities is much harder than processing content. In the DISCERN, 90% of the training time was expended on the IDs. Fortunately, this appears to be a constant factor across the experiments, and not e.g. exponentially growing with complexity. Processing the IDs is hard for two reasons: (1) the system needs to learn to process any ID pattern, i.e. it needs to learn a function, not just how to process a few examples, and (2) the rest of the input pattern provides no cue about what the IDs should be at the output. To produce a correct ID pattern in the output, the network must copy it from the input exactly as it is, without any help from the context.

An interesting comparison can be made to human learning of logical thought [Inhelder and Piaget, 1958; Osherson, 1974]. It seems that children pick up statistical semantics very fast, but are incapable of simple logical inference for a long time. Attributing properties to only specific instances seems to be a hard task for young children.

Maintaining a single, fixed ID for each object cannot be but a first crude approximation of actual identity. In reality, there are several different levels of identities for each object, varying in time and scope. For example, the person John today is not exactly the same as John two years ago, even though at another level John is a unique person over time. What is an appropriate identity depends on the processing context, and the actual boundary between ID and content oriented processing is less well defined and much more complex than in DISCERN.

Elman has addressed this question indirectly in [Elman, 1990; Elman, 1989]. He used a sequential network to read in sentences word by word, with the task of predicting the next item in the sequence. He found that the hidden layer patterns corresponding to different words form intuitively meaningful clusters. Nouns are classified separate from verbs, which are further divided into animate and inanimate subclusters. An important fact about the hidden layer patterns is that they represent not only the word but the whole context. At the bottom of the clustering tree the different occurrences of the word are separated. Elman makes the point that these terminal leaves of the tree represent the instances of the word, i.e. tokens of the same type. The type hierarchy is represented as the non-terminal nodes of the tree, as averages of the instances. Elman's experiments show how the notion of types/tokens can be grounded in the structure of language. The model does not suggest how to make use of the notion in language understanding. The representations, as hidden layer patterns, are internal to the network, whereas in FGREP the representations are globally accessible.

12.6.3 Sequential processing

Recently a lot of work has been done on sequential neural network systems. Being able to deal with sequences is important because in many domains the input or output is naturally sequential, e.g. speech, language, robotic movements, motion detection etc. As discussed in section 4.6.1, sequences also provide means for efficient representation of complex data.

Many sequential architectures are based on Jordan's work [Jordan, 1986]. His model is a three layer backpropagation network, where *the output pattern is fed back* into the input layer, allowing the network to produce sequences. A drawback of this approach is that the network cannot produce sequences where the same output item occurs several times. There is no way to tell which one of the ~~thes~~ in the sequence ~~the boy hit the window with the hammer~~ was just produced.

Elman modified this architecture by using a copy of the previous hidden layer as the source of feedback, i.e. as *a sequence memory* [Elman, 1990]. He has used the architecture to find structure in input sequences, using prediction of the next item as the task. Servan-Schreiber et al. showed that networks of this type could learn to recognize input strings from finite state grammars [Servan-Schreiber et al., 1989].

The recurrent FGREP module is based on the same architecture, but it is used differ-

ently. At input, a sequence of input items is read into a stationary output representation. At output, an output sequence is generated from a stationary input. Because the hidden layer pattern changes after each time step, it is possible to generate output sequences where the same item occurs multiple times. The networks of the FGREP type translate between stationary, wide internal representations of complex knowledge (e.g. thought), and sequential representation of this knowledge, which is used for transmission through narrow communication channels (e.g. language).

12.6.4 Modularity

Most of the PDP models have relied on capabilities of homogeneous architectures, such as simple backpropagation networks. In more complex domains, division into subtasks seems necessary [Minsky, 1985; Ballard, 1987].

For example, Waibel, Sawai and Shikano found that phoneme recognition with homogeneous networks was intractable. However, it was possible to train separate networks to differentiate between subgroups of phonemes, and then combine these networks into larger ones, which with further training could perform the task [Waibel et al., 1988]. In other words, their approach was to *incrementally train and modularly combine*.

Along the same line, Jacobs [Jacobs, 1990] proposes a modular network architecture which learns to decompose the task into functionally independent subtasks. The subnetworks compete to learn the training patterns, and they learn to compute different functions. In effect, the system partitions the input space into regions, and assigns different subnetworks to different regions.

This type of task decomposition is not very useful in high-level cognitive processing. For example the complex task of script processing needs to be broken down into simpler tasks, which are performed sequentially, one task depending on the output of another. The subtasks of a cognitive system may have fundamentally differing characters, such as storing items in the episodic memory vs. computing a function.

DISCERN aims at the latter kind of modularity. The processing emerges from cooperating hierarchically organized modules, as well as episodic memory and lexicon modules. The model combines fundamentally different processing architectures in order to accomplish a high-level task. The modules execute well-defined subtasks, and communicate directly with explicit and meaningful internal representations.

The symbolic/connectionist script applier system of Lee et al. [Lee et al., 1989] consists of separate networks for developing word and event representations, learning sequences of events, and associating roles with fillers. The model goes beyond modularity in neural networks, in that the interactions of the modules are mediated by a higher level symbolic system. The subnetworks operate independently as *PDP modules in a large symbolic system*. This is different from DISCERN, which is an integrated neural network model.

12.6.5 One-shot storage in associative memory

The storage mechanism in the episodic memory essentially has to implement a content-addressable associative memory. Several patterns need to be stored in the same memory, with similar patterns interfering with each other. The performance should degrade gracefully when the memory is overloaded. The items need to be retrieved with partial or approximate patterns as a cue.

However, the memory must perform learning which is not possible in most models of associative memory. The patterns to be learned are continuously valued, non-orthogonal and not necessarily linearly independent. Each item needs to be stored one-shot, with only a single presentation. This must be done without knowledge of future items to be stored, and without re-activating the traces of the previous items.

Most parallel models of associative memory require that all patterns to be stored are known in advance. Usually the patterns are learned incrementally, by cycling through the set of all patterns multiple times and modifying the network weights slightly at each presentation with a local learning rule. Eventually the weights converge to a configuration where the network makes the right associations to all patterns in the training set, and possibly generalizes to new but similar patterns. Backpropagation networks, Boltzman machine [Ackley et al., 1985] and the Brain-State-In-A-Box model [Anderson and Mozer, 1981] are examples of this approach.

The above models can learn patterns with a single presentation only when the patterns are orthogonal. In this case there is no crosstalk between patterns, and the learning rates can be made large enough to code the patterns completely in a single modification. If the patterns are not orthogonal, each new presentation affects the coding of previous patterns, and the early memories are gradually wiped out. Several presentations of each with very small modifications are necessary for the process.

In some special cases, e.g. with linearly independent binary patterns, it is possible to determine the appropriate weight values with a non-local algorithm, and no learning is necessary [Kohonen, 1984]. Still, all the patterns need to be known in advance, and new patterns can be added only by recomputing the whole weight matrix from the complete set of patterns. The Hopfield network [Hopfield, 1982; Hopfield, 1984] is perhaps the best-known example of this type of associative memory.

One-shot learning in the above models is infeasible because they distribute the trace of each item over the whole memory hardware, and crosstalk cannot be avoided. The solution to this problem is to use different hardware to store different memories, i.e. to add a degree of localization to the memory representation.

This idea is employed in e.g. Nenov, Read, Halgren and Dyer's model of auto-associative memory, which is based on the Gardner-Medwin model of the hippocampal formation [Nenov et al., 1990]. Each item is represented by a binary activity pattern over the network. In each pattern, approximately 10% of the neurons are active, and these neurons are spread randomly over the network. Each neuron is randomly connected to about 6% of the other neurons in the network. Initially the connections are ineffective with 0 weight. A trace is created through Hebbian learning, i.e. by increasing the weights between active neurons

in the pattern to 1. Because only 10% of the neurons are active in any given pattern, the different memory traces are coded largely in separate hardware. It is possible to store patterns in this memory one at a time, with little crosstalk. In retrieval, a cue pattern is presented and the network activity settles to a stable pattern. Incomplete and noisy cues can be completed, and the model indicates no-recall by settling to an inactive state. Spurious recall can be controlled by modulating the firing thresholds through tonal and recurrent inhibition.

Kanerva's sparse distributed memory [Kanerva, 1988] is another associative memory model for high-dimensional binary patterns capable of one-shot storage. Each neuron in this model is a high-dimensional address decoder, and contains storage space for several vectors of the same dimensionality. In effect the address space is the data space. The memory is sparse in that only a few of the 2^n addresses are coded in the neurons, which number far less than 2^n . However, each vector is stored at a number of neurons, the ones whose addresses are closest to this vector. Each neuron participates in storing several vectors. When an incomplete cue is presented, a number of neurons whose addresses are close to it are activated, and the average of all vectors stored in these neurons is used as the next approximation of the vector to be retrieved. The process is repeated, and if the original cue is sufficiently close, the process converges to the closest stored vector. Interestingly in this model, even when the one-shot storage without crosstalk is provided by the structure of the neuron, storing the vectors at specific parts of the hardware is necessary for the retrieval process.

The trace feature maps are capable of one-shot storage and associative retrieval like the above models, but they have two additional features: (1) the hardware that participates in the storage of an item forms a spatially localized area of the network, and (2) the storage patterns can be continuously valued.

12.6.6 Role binding

Role binding (or more generally, variable binding) is one of the central mechanisms in cognitive modeling, and one of the hardest to implement in connectionist systems. It arises from the need to represent general processing knowledge, which can be applied to several specific situations. For example, the general knowledge about restaurant visits is independent from the actual events in a particular visit. The frame is the same, the fillers change from one visit to another.

Our approach is to use data-specific assemblies for the roles, with distributed patterns in the assemblies indicating the fillers. Role-specific assemblies would be adequate for e.g. sentence processing, because the number of roles is small and remains fixed. The situation for representing roles for different scripts is more complicated, because the roles are different from script to script. It was necessary to use base assemblies to indicate what roles the other assemblies stand for. This is a simple yet powerful mechanism, and combined with the ID+content technique, it will adequately do the task of role binding in scripts.

It is possible to go even further and use distributed representations for the roles also. Binding can be represented as a tensor product of the two representation vectors [Smolensky.

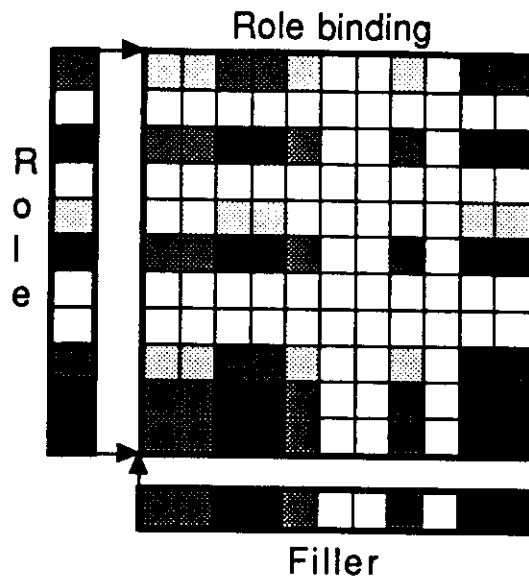


Figure 12.4: **Representing role binding in a tensor product network.** The values of individual units (indicated by gray scales) correspond to the tensor product of the representation vectors, i.e. $t_{ij} = f_i * r_j$, where t_{ij} is the role binding unit, f_i is the filler unit, and r_j is the role unit.

1987; Dolan, 1989; Dolan and Smolensky, 1989]. The tensor can be implemented as a square network of processing units, where the unit activities correspond to the values of the tensor components (figure 12.4). A role representation can be recovered from this network by specifying the filler, and vice versa. The set of roles does not have to be specified in advance, the number of roles and their representations are totally open. Several bindings can be superimposed on the same network by simply adding the tensors. However, crosstalk is a serious problem, and some type of cleanup processing is usually needed to improve the accuracy of the retrieved vector [Dolan, 1989].

Smolensky has shown that many of the role binding techniques suggested so far can be mathematically analyzed using the tensor product approach [Smolensky, 1987]. Tensor products are a special case of conjunctive coding. In e.g. the DUCS system [Touretzky, 1987], the role-filler pairs are represented in a matrix of units conjunctively. Derthick's μ KLONE system uses a similar encoding for roles and fillers to represent knowledge about individuals [Derthick, 1988].

The tensor product representation has been extensively explored in Dolan's CRAM system [Dolan, 1989]. The top level of CRAM is a symbolic schema-based natural language understanding system. The knowledge operations have been implemented with connectionist modules at the low level. The schemata, their roles and fillers are represented distributively, and the schema instantiation and role binding are implemented in terms of tensor operations on these representations. CRAM performs like a symbolic model at the high-level, but it is able to make use of the robust processing features of the distributed modules in schema instantiation.

The symbolic high-level control in CRAM makes it possible to do three types of variable binding: pattern-match (as in DCPS), constraint propagation (as in μ KLONE) and expectation based binding (as in DISCERN). In other words, CRAM can use statistical regularities for the binding, but it can also perform novel bindings, which is hard to do in purely distributed systems (section 11.8).

12.6.7 Parallel distributed inference

Inference is often modeled in artificial intelligence systems based on probabilistic formalisms ([Pearl, 1988; Duda et al., 1977; Buchanan and Shortliffe, 1985]). Events are known with certain probabilities and they provide evidence for other events. The dependencies are well-defined, and inferences are drawn using conditional probabilities. Sound estimates of the certainty of the inferences are obtained, and coherent belief systems can be maintained. Unfortunately, this level of accuracy and rigor is very hard to achieve in high-level cognitive modeling, as in natural language understanding. The conditional probabilities are not well defined and dependencies are very complex and fuzzy. *Finding the relevant data is the main problem.* On the other hand, obtaining accurate measures of the reliability of the inference is not crucial. The task is to find the most relevant data from an enormous collection of information, and build a hypothesis based on that, knowing that this is only an approximation of the best inference that could be drawn mathematically.

It seems that people have two fundamentally different mechanisms at their disposal for inferencing. The relevant data can be searched for using a sequential symbolic strategy. One does not have an immediate answer to the question, but the answer is sequentially constructed from stored knowledge by a high-level goal directed process, i.e. by reasoning. Another type of inference occurs through associations immediately, in parallel, and without conscious control, i.e. by intuition. Large amounts of information, which may be incomplete or even conflicting, are simultaneously brought together to produce the most likely answer.

Neural network systems fit well into modeling intuitive inference (see also [Touretzky, 1989a]). The network extracts statistical knowledge from the input data, and these statistics are brought together to generate the inference. For example, the amount of the tip in the restaurant story is inferred from the whole story, not just by looking at some part and applying a specific rule. If the food was bad, the network usually infers that the tip was small, but if the customer ate a hamburger, the representation in the tip slot will be closer to no-tip, because hamburgers are usually eaten in fast-food restaurants. In other words, neural networks are able to perform probabilistic inference, not by coming up with a specific answer and its probability, but by producing an answer which is the average of all alternatives, weighted by their probabilities. This is exactly what is needed in e.g. filling in the missing events and fillers in a script-based story. On the other hand, this type of inference is not useful if we are trying to understand unusual events [Touretzky, 1989a; St. John, 1990].

Touretzky has recently proposed how connectionist systems could perform more complex *reasoning*, by exploiting compositionality of data [Touretzky, 1989a]. For example in language, meanings of complex inputs are composed of meanings of simple ones, and an exponential number of internal states could be produced even by a system which is trained

with only simple inputs. He points out that no such systems, named “dynamic inferencers” by him, has yet been built within parallel distributed paradigm.

The distributed connectionist production system [Touretzky and Hinton, 1986] is probably furthest in this direction. The execution of this system is based on rules, represented distributively in a separate memory, operating on data triples in the working memory. The rules consist of two clauses, which are matched against the working memory, and the right hand side which consists of commands for adding or deleting triples from the working memory. A typical rule could be e.g. $[(x A B), (x C D) \rightarrow \text{add}(G x P), \text{del}(x R x)]$, where the lower case x represents a variable. The rule would fire if the triples $(F A B)$ and $(F C D)$ exist in the working memory, adding $(G F P)$ and removing $(F R F)$ from the memory. Thus the system has an exponential number of working memory states and combines the components of the triples to construct new states.

DISCERN is not designed to model inference with explicit rules. However, one essential constituent of dynamic inference is variable binding, and the mechanism of IDs is a step toward this direction. As discussed before in section 13.3.1, the DISCERN system can process a combinatorial number of inputs with linear cost.

Chapter 13

Extensions and future work

A major contribution of DISCERN is to show that building a complete natural language processing system from distributed neural networks is feasible. DISCERN can provide a framework for future research in building complex modular neural network systems, and for research in distributed language processing, episodic memory, lexicon, semantic representation and control. Much of these tasks have been finessed in DISCERN, and they call for further research. On the other hand, DISCERN shows how inadequate simple low-level techniques are in processing complex structures and novel situations. Representing and learning knowledge structures appears to be the most important research problem on our way towards artificial neural network intelligence.

13.1 Processing more complex texts

13.1.1 Pronoun reference

Pronoun reference is not a particularly hard problem in understanding script-based stories. People do not get confused when reading e.g.: **The waiter seated John. He asked him for lobster. He ate the lobster.** The events in the story are stereotypical and once the role binding has been done, the reference of the pronoun is unambiguous. It should be possible to train the network to deal with pronouns in the stories. Some of the occurrences of the referents can be replaced by **he**, **she** or **it**, and very likely the representation for these words will develop into a general actor, food etc.

13.1.2 Multiple scripts

It might be possible to develop a mechanism for representing multiple scripts and their interactions (e.g. a telephone script or robbery script occurring within a restaurant script). One approach would be to use distributed representations for the roles and the scripts, instead of fixed, designated assemblies [Dolan and Smolensky, 1989; Dolan, 1989]. Instantiation of a script would now have the form of a tensor product "cube". One face of the cube would stand for the script and track, one for the role, and one for the filler. It should be possible to represent multiple simultaneously active scripts in the same cube. Scripts could be partially activated and their boundaries would be less rigid.

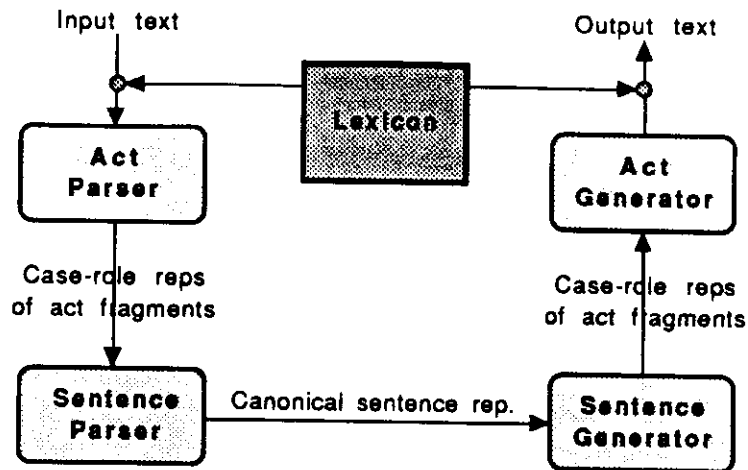


Figure 13.1: Block diagram of a system for parsing and paraphrasing sentences with relative clauses. The model consists of parsing and generating subsystems, and a central lexicon of distributed word representations. Each subsystem consists of two hierarchically organized modules, with the case-role assignment of the act fragment as an intermediate representation. The complete sentence representation is a concatenation of the complete case-role representations of the acts in the sentence.

13.1.3 More complex syntax

DISCERN currently processes syntactically very limited language. The sentences consist of single clauses and the number and types of predicates are limited by the available case roles. Some extensions to more complex syntax are feasible. A separate system which reads and paraphrases sentences with multiple hierarchical relative clauses has already been implemented [Miikkulainen, 1990b]. Combining this architecture with DISCERN would be an interesting extension.

Instead of simple sentence parser and generator, the sentence processing in [Miikkulainen, 1990b] is performed by a hierarchical organization of two modules very similar to the parser and generator modules in DISCERN (figure 13.1). The act parser module reads the input sentences one word at a time, and forms a case-role representations of each act fragment as its output (figure 13.2). An act fragment is any part of the sentence separated by commas. For example, in reading *The woman, who helped the girl, who the boy hit, blamed the man*, the first act fragment is *The woman*, and its case-role representation only consists of *agent=woman*.

The sentence parser in this system receives a sequence of act fragment case-role representations as its input. Its task is to keep track of the recursive relations of the act fragments and combine them into complete act representations. It also ties the different acts together by determining the referents of the relative pronouns. At the output of the sentence parser, the act representations are concatenated into a single vector.

The sentence parser output is a canonical representation of the sentence in terms of its acts. The recursive clause structure is not explicitly represented, but it is implicit in the act

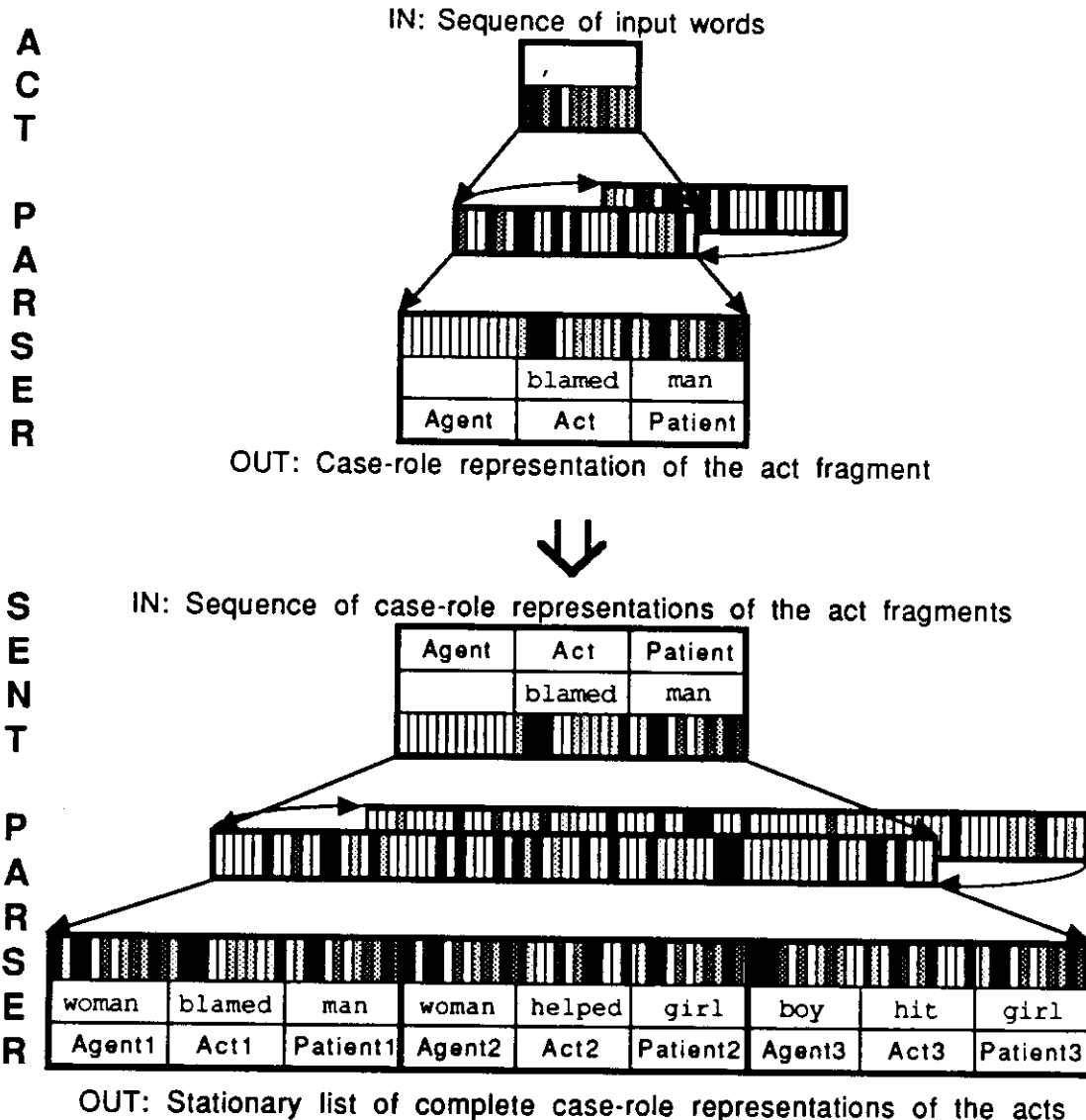


Figure 13.2: **Networks parsing the relative clause sentence.** Snapshot of the simulation after the whole sentence **The woman, who helped the girl, who the boy hit, blamed the man** has been read in. The output of the act parser shows the case-role representation of the last act fragment, **blamed the man**. The output of the sentence parser displays the result of the parse, the internal representation of the whole sentence. The generator networks reverse the parsing process. A snapshot of the generation process looks almost exactly the same, only the arrows are reversed and the hidden layer patterns are different.

representations, e.g. two acts sharing the same agent, the agent of one act is the patient of another etc. The point is that *the recursive structure is a property of the language, not the information itself*. Internally, the information is represented in a parallel, canonical form, which makes all information directly accessible. In communication through narrow channels, i.e. in language, it is necessary to transform the knowledge into a sequential form (section 4.6.1). Parallel dependencies in the information are then coded with hierarchical relative clause structure.

For example, in the above sentence there were three acts, each consisting of an agent, a verb and a patient. The sentence parser forms the vector |woman blamed man| |woman helped girl| |boy hit girl| as its output (the words indicate semantic representation patterns and the bars indicate case-role representation boundaries, included for legibility).

The sentence and act generators reverse the parsing process. The sentence generator produces a sequence of case role representations of the act fragments, and the act generator outputs the representations for the words in each fragment. Depending on the training of the sentence generator, the same sentence can be paraphrased using different syntactic structures. The above sentence can be expressed with hierarchical relative clauses: **The woman, who helped the girl, who the boy hit, blamed the man.** Or, it can be expressed as a sequence of simple clauses: **The boy hit the girl, the woman helped the girl, the woman blamed the man.**

The connection weights of the sentence generator determine the style of the output. It would be possible to add a higher-level decision-making network to the system, which controls the connection weight values in the sentence generator network through multiplicative connections [Pollack, 1987]. A decision about the style, detail etc. of the paraphrase would be made by this module, and its output would assign the appropriate function to the sentence generator. The system would adjust its output style according to the context. A similar mechanism could be used to generate paraphrases in different languages.

13.1.4 Translation

The lexical symbols and the language interface modules (the sentence parser and the sentence generator) are independent from the rest of DISCERN. It would be possible to train these modules to read and produce output sentences in different languages.

The semantic map of the lexicon is, in principle, language independent. It could be attached to several symbol maps in different languages. A global parameter would determine which symbol map is being used. For example, the activity from the semantic unit **boy** would propagate to **BOY** on the English symbol map, or to unit **JUNGE** in the German symbol map. Different versions of the sentence parser and sentence generator could be trained to translate between the case-role representation and the sequence of meanings in different languages. For example, **John ate a good steak** would be translated into **John ate a steak good for Spanish**, where the adjective follows the noun.

However, words in different languages often have slightly different meanings, and the above system could only do very simplistic translation. There are two ways to deal with the different meanings: (1) All word meanings from different languages could be represented on

the same semantic map. Any meaning could be used for the internal representations. Some units would have connections only to English symbol map, others only to German map. Or, (2) the semantic representations of each language could be represented in different maps. Again, all these maps could be accessible for internal representation, or one of the maps could be primary, i.e. a "native language map".

Interesting questions arise from this organization. A bilingual system could use representations from two different language maps for internal representation. Alternative (2) implies that in localized brain damage, it would be possible to lose access to words and meanings in one language, while preserving the corresponding words in another language. This alternative also favors complete code-switching between languages. In alternative (1), if the category-specific impairment occurs, it occurs in all languages. Slips between languages could be easily modeled.

13.2 Semantic representations

13.2.1 Cumulative representations

The FGREP representations adapt to the task and data. If the training data is changed, the representations will adapt, and reflect the new requirements. What was previously learned will gradually fade away. In some applications this is appropriate, but in others (e.g. in learning the meaning of words) the effect should be mostly cumulative. The old information should become gradually more rigid.

One possible way to achieve this is to use weights with different learning rates [Hinton and Plaut, 1987]. Another possibility would be to let the size of the representation grow as more data comes in. The fastest learning would occur in the new area, while older areas changed slower. New information would be learned in terms of what is already known. If the input is familiar, the representation would be changed within the current area and would interfere with the previously learned. If the input is very different from the previous ones, new units would be acquired to prevent unlearning previous information. Periodically the representation could be reorganized to use the units more efficiently.

13.2.2 Composite representations

Knowledge about the use of the words alone cannot constitute a complete semantic system. If every word in our system is defined only in terms of other words, the system has no grounding. Some words also need to have sensory content of their own, based on their referent.

The IDs can be seen as approximating such sensory content. These could be distributed representations of visual images, obtained through the sensory pathway to the lexicon (section 9.5). The representation for a concept would have two parts: sensory and linguistic.

It is possible to take this idea even further, and make the concept representations more compositional. A concept is a composite of several representations from different points of view. E.g. John would have a linguistic representation which codes how this word is used

in language, a visual representation of the particular person, representation of this person over different time periods etc. The relevant parts of the composite representation would be pulled out for each task.

13.3 Lexicon

13.3.1 Extending the lexicon

DISCERN could extend its training lexicon dynamically when needed. Each time a new word is encountered in the training data, the training supervisor would create a new entry for it in the lexicon. Expectations generated by the network could be used to come up with an initial guess for the representation. The expectation pattern in the appropriate slot (specified by the training data) is first matched against the lexicon. If there are no words with a similar representation, the expectation itself could be used as the initial pattern for the word. If there is a prototype close to the expectation pattern, a new instance of that prototype would be created and assigned to the new word. Because the expectation for **MaMaison** (as a new word) would probably be very much like **fancy-restaurant**, the initial lexicon entry for **MaMaison** would be an instance of **fancy-restaurant**.

The initial entry for the new word would be modified through experience, and it would acquire semantic content of its own. The instance would gradually become a new prototype itself. Eventually there would be several different fancy restaurant names in the vocabulary, each with an established unique meaning. The original fancy-restaurant representation would still be in the lexicon, representing a generalized form of these specific instances. This mechanism of learning new words could model forming of subclass hierarchies, i.e. more specific terms from general terms.

The lexicon in DISCERN (both during training and during performance) is primarily a model of single word processing. It does not have special mechanisms for representing and processing phrasal structures and morphology. There are two possible ways of doing this, and it seems that both of them are involved. Common morphological forms and phrases, such as **nationalism** or **The Big Apple** could be represented like words, as single entries. More complex phrases and unusual, constructive forms, e.g. **kick the bucket** or **non-preemptive** could be represented in the lexicon by their constituents, and parsed/generated by a higher-level language processing module.

13.3.2 Combining FGREP with the lexicon

Currently, the FGREP process is separate from the double feature map model of the lexicon. The external training supervisor maintains the semantic representations during FGREP. The actual lexicon is trained only after the representations have been developed and all word instance representations have been created from the prototype representations by assigning ID patterns. During performance, no external supervision is needed. The semantic representations have been stored in the semantic map, and the lexicon now maintains them independently.

In effect, the FGREP lexicon is an abstraction of the performance lexicon. From the engineering point of view this makes no difference, since the connections between modules are severed during training anyway, and DISCERN does not function as an independent system until performance. However, if DISCERN is viewed as a learning model, it would be desirable to use the actual lexicon module in the FGREP process.

The current representations would be stored in the weights of the semantic map units, and they would be initially random, corresponding to the random initial organization of the lexicon. The representations would be fetched from the lexicon by presenting the lexical word to the lexical map and propagating the activity to the semantic map through the associative connections. Changing the representations with FGREP would now mean actually changing the input weights of the unit. The lexical and semantic maps and the associative connections would be organized simultaneously with developing the representations.

The lexicon would now need to be trained with random ID patterns. As was discussed in section 9.4.1, finding appropriate symbols to associate the random instances with becomes a problem. Also, nothing in the above process prevents the lexical \rightarrow semantic associative connections from converging onto a single semantic unit. An additional mechanism would be needed to force the process to spread out the associations over the whole semantic map. These are currently open research problems in combining FGREP with the lexicon architecture.

13.3.3 Disambiguation

Priming and disambiguation are currently not implemented in the lexicon module of DISCERN. The story data that DISCERN was trained and tested on did not contain ambiguous words, although synonyms and homonyms are quite common in everyday language. The lexicon was designed specifically to model the many-to-many mapping between symbols and their meanings, and it should be possible to make better use of the lexicon architecture in DISCERN. A possible mechanism for disambiguation was outlined in section 9.3.5. The associative activation in the lexicon is combined with expectation activation, generated by the sentence parser. The largest total activity wins, i.e. the alternative which matches the expectations is selected as the meaning of the lexical symbol.

The expectation activity could be obtained from the output of the current sentence parser, but this requires deciding which of the ambiguous slots is likely to be read next. Perhaps the simplest way is to train the sentence parser also to predict the next word in the sequence (in addition to forming the case-role representation). The prediction task is well-suited for simple recurrent networks [Elman, 1990; Small, 1990], and should be possible to do without taxing the performance of the sentence parser very much. The prediction output could be directly used to prime the lexicon.

Currently, the lexicon simply selects and outputs the representation stored at the maximally responding unit. The selection could also be implemented with lateral inhibition, where the map settles into a localized response around the maximally responding unit [Miikkulainen, 1987]. The settling times should be similar to the reaction times observed in humans [Simpson and Burgess, 1985]. High-frequency words would have shorter settling

times (with more focused associative connections), and these times could be changed with priming. With several equally likely interpretations, settling would take longer.

13.4 Episodic memory

13.4.1 Extensions to hierarchical feature maps

The representation of role bindings at the bottom-level maps of the episodic memory at first glance seems to suffer from combinatorial explosion. The number of different binding combinations that the system can tell apart is limited by the number of units in the bottom-level maps. If k is the number of input dimensions and N is the number of different values in each dimension, N^k units are needed to represent the space of all possible combinations. This " N^k problem" is a general limitation of the value-unit encoding approach [Allman et al., 1981; Ballard, 1986].

A plausible solution, which also seems to be in use in the value-unit maps in the central nervous system, is to divide complex spaces into lower-dimensional spaces, and combine them hierarchically [Ballard, 1986]. None of the role-binding maps should ever need to map more than a few dimensions. If there are many dimensions of variation, these could be split into separate maps. In other words, the hierarchy of maps would have two kinds of hierarchical relations: (1) a higher level map acts as a filter, forming categories of the space and passing down subcategories to a more accurate mapping at the lower level, and (2) the whole input space of the parent map is laid out at the lower level, but different dimensions are mapped on different submaps. The representation of an item in such a hierarchical system would be an AND-OR tree. In the type (1) submapping, only one submap is active for each input item, whereas in the type (2) mapping, the representation consists of image units in each and every submap.

It might be also possible to develop a mechanism for automatically adjusting the sizes of the maps or the depth of the hierarchy according to the input data. Two different inputs should always be mapped onto different units at the lowest level. This constraint could be used to self-organize the system architecture, in addition to the current self-organization of the individual feature maps.

The map organization also seems to waste hardware. Whole feature maps below intermediate units are never used. This is more a property of the simple implementation than the process itself. The hardware could be allocated where it is needed. Units that never receive inputs could become part of active maps, improving the accuracy of this map. This could be part of the process of self-organizing the system architecture.

Currently, the number of role bindings that can be represented is limited by the number of units on the bottom level maps. The resolution could be improved by taking the response pattern as a whole into account. Instead of using the input weights of the maximally responding unit as the representation, the weight vectors of all active units could be combined, proportional to their activity. Role bindings "between" units on the map would be represented as linear combinations of existing unit weights.

As discussed in section 6.1, the theory of feature maps is based on complete response

patterns of very large maps. Taking the maximally responding unit for the image was a convenient shortcut, and quite enough for most self-organization tasks. However, it seems that many interesting phenomena on feature maps depend on more accurate modeling of the biologically plausible mechanisms.

For example, developing the associative connections in the lexicon is possible only with localized response patterns on the lexical and semantic maps. It was also necessary to pass the activity through the most immediate neighborhood of the maximally responding unit in order to filter out inaccuracies in the input representation. In the trace feature maps, the retrieval requires modeling the settling process itself. Going back to the response patterns and the settling process makes the self-organization process much harder to control and analyze, but it also opens possibilities that go far beyond the capabilities of current feature map systems.

13.4.2 Extensions to the trace feature map mechanism

The trace feature map model has several interesting properties which have not been pursued further in DISCERN, but constitute possible directions for future research.

The global parameter θ (equation 8.1) is used to select between storage and retrieval processes. When $\theta = 0$, the lateral connections of the map are inactive, and the map develops a response according to the external input activity only. This response is coded into the lateral connections as the memory trace. With $\theta > 0$, the lateral connections are active, and the map settles to a stored trace. Lateral connections are not modified in this case.

Interesting effects would result from relaxing this simple two-mode operation. If lateral connections remain active during storing of an item, they will affect the trace that is formed. Items that are novel, i.e. in a sparse area of the map, will be stored as before. Items close to existing traces will be pulled closer to the traces. In effect, the input pattern is perceived as more similar to the existing traces than it actually is. Contents of the memory affect how new items are remembered.

On the other hand, it is also possible to modify the lateral weights during recall. After the activity has settled, the lateral connections can be changed within the final activity pattern. If the initial activity is emphasized in the settling process (with small θ in equation 8.1), the final pattern reflects the initial activity more than the lateral trace connections, and the trace is changed towards the question. This has the effect of questions modifying the memory, which is psychologically valid behaviour [Loftus, 1975].

The settling times give interesting insight into the recall process. Settling usually takes a few iterations longer when the cue is far away in the map. When there are two traces about as far from the cue, convergence can take very long. In other words, retrieval from an ambiguous cue is slower than retrieval of a unique item. The settling times provide information that could be used by a high-level monitor process to decide on the validity of the recalled trace.

Usually one of the ambiguous traces wins after a number of iterations, turning off the other one completely. However, it is also possible that both traces remain active in the

stable pattern, indicating ambiguous retrieval. Our implementation currently retrieves the one with the strongest activity. It would also be possible to retrieve an average of the two traces. This way it would be possible to model situations where a blend of two traces is retrieved, e.g. **blue circle** from **red circle**, **blue square**.

Modulating the width of the traces has not been explored at all in the current model. It would be possible to make the early traces more resistant to forgetting by storing them on a wider area. It would be possible to model the primacy effect on memory recall. Also, traces on a crowded section of memory could be made smaller, and more of them would fit in without severe interference.

Interestingly, decreasing the width of the trace with experience is in line with the self-organizing process, where the neighborhoods initially are large but decrease as the map becomes ordered. Perhaps traces could be created during self-organization also. Or, traces could be part of self-organization, modifying the lateral connections to better support the process. These are completely open questions at this point.

13.5 Question answering

Question answering is not very complicated in DISCERN. The main goal is to demonstrate scrip-based inference. Question answering in general is a complex process [Lehnert, 1978], and a very interesting area for future development of DISCERN.

The trace is not modified during retrieval in DISCERN. If the cue is close enough to a stored trace, the trace is always returned as it is. However, questions do modify memory [Dyer, 1983]. They can supply more information, which becomes part of the memory trace, or questions with incorrect assumptions (leading questions) can change part of the trace [Loftus, 1975].

A possible way to modify DISCERN so that the trace is affected by retrieval was discussed above in section 13.4.2. However, additional mechanisms are necessary to guarantee that the change makes sense. Both the cue and the trace may contain uncertain information, but it is hard to see where it is. The trace is always stored as a complete representation vector, even if DISCERN had to guess parts of it. Where the question representation differs from the trace, it may be incomplete (to be completed by the trace), incorrect (to be corrected by the trace) or it may specify new information (to replace the trace).

Currently there is no way of telling which parts of the representation are certain and which are just guesses. The system should somehow represent this information, and the changes should be based on the relative certainties of the representations. The incomplete parts of the question should not affect the trace, whereas the incomplete parts of the trace should be made vulnerable to the question effects. How this could be done is currently an open research question.

Often ambiguous questions can be resolved by taking the context of discourse into account [Lehnert, 1978]. For example, after asking **Did Mary travel a big distance?** it is quite possible to ask simply **Where?**, and it is obvious that the question refers to Mary's travel destination.

Currently DISCERN processes each question independently from the previous ones. The cue is formed and the appropriate trace is retrieved for each question in isolation. However, it would be possible to model context effects with high-level control. The previous story representation could remain in the immediate memory, and only if the question cannot be answered with it, the cue is presented to the episodic memory. Or, the retrieval could be performed for each question, but residual activity could remain on the episodic memory feature maps. This activity would prime the previous story in the retrieval for a highly ambiguous cue.

13.6 Implementing control with networks

In our simulations an external symbolic system took care of the control of execution. Nothing very complicated is involved in it, and control could well be implemented with simple networks.

In the performance phase of DISCERN, input and output segmentation is based on the period at the end of each sentence. Special networks could be trained to recognize the period (and the question mark), and control the gateways with multiplicative connections [Rumelhart et al., 1986a; Pollack, 1987]. As soon as the sentence parser has read the period, one such gating network could open the pathway from the output of the sentence parser to the input of the story parser, which can then execute one output cycle. Similarly, another gating network would recognize the period at the output of the sentence generator, and send a signal back to the story generator. This signal would open the pathway from the hidden layer to the previous hidden layer, allowing the network to produce the next case-role representation.

More generally, the control networks could be implemented as additional modules, trained in the control task as any other modules in the system. The control module would receive input from several pathways in the system, and its output would gate the pathways through multiplicative connections. Interesting issues in generalization, expectations and robustness of control make this an interesting research direction.

The different modules of DISCERN have capabilities which are not used in the complete DISCERN model. For example, the lexicon can disambiguate word meanings with priming activation, the episodic memory can detect multiple possible traces and indicate when there is nothing appropriate in the memory. The system can make plausible guesses about missing information and correct inaccurate questions, and it could also modify the trace when the question supplies new information. However, making use of these capabilities requires monitoring the system performance at a higher level. Such high-level control does not exist in DISCERN at the moment.

There are three main tasks for a high-level monitor in DISCERN:

(1) It could detect and correct errors which cannot be caught by the automatic low-level filters, such as confusions between IDs.

(2) The high-level monitor could control ambiguous retrieval. It would decide where to look for a trace, when it is appropriate to correct errors in the question and when it should modify the trace according to new information in the question, how to respond when multiple

traces are found, or when nothing is found (see section 8.3.3).

(2) Currently, DISCERN tries to process every input exactly the same way, whether the input makes sense or not. The monitor process could recognize situations where the system is producing garbage or making too many guesses. It would monitor the feasibility of the task and the quality of output, and stop the execution if it makes no sense.

These three tasks are important in making DISCERN perform robustly in more real-world-like situations. It might be possible to implement some of the necessary monitoring, decision making and modulation of low-level processes with additional control networks, as outlined above. Eventually, the goal would be to develop distributed control modules which would act as a high-level "conscious" monitor for DISCERN.

13.7 Discovering scripts

The learning of scripts in FGREP modules is based on a slot-filler representation of the story. An external supervisor has to decide on a fixed set of slots (such as customer, food, who-to-pay etc.), and tell the system what the correct fillers should be for each story. The disadvantages of this approach are (1) the number of different scripts the system can represent is limited by the slot-representation, and (2) the slot-representation must be designed in advance, which means that all the variations of all stories must be known in advance. Even when the FGREP modules are extracting the regularities that make up the scripts, the possible scripts are implicitly determined in designing the training data.

The hierarchical feature map approach does not have these limitations. The breakdown of each script is independent of the others and not limited by a fixed assembly-based representation. Representations for scripts, tracks and roles are developed automatically according to the data. The stories are classified according to the statistical similarities in the input representations, with no preconceptions about what the classification is going to be. The only constraint is that the classification should facilitate the best discrimination between the instantiations of the script. The breakdown of a script into tracks and role bindings is done independently of other scripts, and is usually different for different scripts.

On the other hand, the hierarchical feature map is a data organization, not a processing model. For processing we need something like the FGREP network, which can be trained to perform a particular task. An interesting question is, would it be possible to use the self-organizing system to determine the regularities in the data, and then use the information it has found to automatically generate training data for supervised-learning based higher-level systems?

Learning script representations is in general considered a hard task [Rumelhart et al., 1986d; Schank, 1981]. However, scripts are regular event sequences, and it should be possible to learn these structures and their hierarchical relationships from the experience. This is how people develop new schemata. Scripts provide a means to organize experience.

It has been shown [Miikkulainen, in press], that hierarchical feature maps can extract the script taxonomy also when the example stories are represented directly by a concatenation of their sentences. Perhaps the taxonomic knowledge, i.e. the similarities and differences

that the taxonomy is based on, could then be extracted from the feature map system, and be used to form internal representations for the stories. This representation could in turn be used to train FGREP networks to process the actual stories. This approach would combine the advantages of both unsupervised and supervised learning.

13.8 Representing and learning knowledge structures

The sentence processing system discussed in section 13.1.3 can learn to parse and paraphrase fairly complex sentences with multiple hierarchical relative clauses. However, it does not develop an abstract representation for the clause structure, and it cannot do very well with novel syntactic constructs.

As discussed in section 11.8, inability to deal with novel inputs is a fundamental limitation of current distributed models [Touretzky, 1989a; St. John, 1990]. These models are activity pattern transformers, and they must be specifically trained for each different type of input. For example, the sentence processing system needs to see all different combinations of relative clauses in the training set. If it is trained on "A, who did B to C, did D to E", it cannot read an additional clause "...E, who did F to G" [Miikkulainen, 1990b]. It has no representation for the *general structure* of a relative clause, it can only process activity patterns which are similar to the ones it has seen before.

On the other hand, architectures have been proposed which learn to recognize strings from some potentially infinite grammar [Servan-Schreiber et al., 1989; Stolcke, 1990]. They are state machines which have the knowledge about the grammar but have no mechanism to represent the result of the parse.

An important direction of future research is to develop methods for representing and learning abstract structures that would make it possible to generalize into novel situations. For example, if the sentence processing system could represent the general structure of relative clause, it could apply that structure to novel clause constructs.

Inseparable from the question of how one represents structure is how one can learn structure from experience. Ideally, the system would extract the structure from the input examples and use it to represent the inputs. The structures are regularities in an abstract level, and they provide an efficient way to organize the examples. It should be possible to extract the regularities through self-organization. Extended into higher-level descriptions of the data, the self-organizing process should develop representations which efficiently encode the higher-level regularities. The actual methods and architectures for doing this are future research.

Chapter 14

Conclusions

14.1 Summary

DISCERN is a large-scale natural language processing model built entirely from distributed artificial neural networks (figure 14.1). *Modularity* is a key concept in the DISCERN architecture. The different script processing subtasks and the lexical, semantic and episodic memory components are realized in separate modules. The modules are trained separately with compatible training data. Processing modules are feedforward and simple recurrent networks, trained with backpropagation. The memory modules are implemented as feature maps, organized in an unsupervised learning process.

The I/O of the subnetworks consists of distributed representations stored in a *central lexicon*, and the modules communicate using these representations. The lexicon representations are developed automatically by all processing networks while they are learning their processing tasks. With backward error propagation extended to the input layer, the representations are developed by the *FGREP mechanism* as if they were an extra layer of weights, while at the same time making them *publicly available* in the lexicon. FGREP creates a *reactive training environment*, i.e. the required input/output mappings change as the I/O representations change.

Single units in the resulting representations do not stand for clearly identifiable semantic features or label distinct categories. All aspects of an input item are distributed over the whole set of units in a *holographic* fashion, making the system particularly robust against noise and damage. Each representation also carries *expectations* about its possible contexts. The representations evolve to improve the system's performance in the processing task and therefore efficiently code the underlying relations relevant to the task. This results in extremely good generalization capabilities, superior to PDP systems with semantic feature encoded representations.

The lexicon can be extended by cloning new instances of the items, i.e. generating a number of items with the same properties but with distinct identities. This is accomplished by combining the semantic representation with a unique ID-representation. This *ID+content technique* is motivated by sensory grounding of words, and forms a basis for symbolic processing in PDP. It is possible to approximate a large number of items by dividing them into equivalence classes, resulting in combinatorial processing capabilities with linear cost.

Representing input/output as *sequences* overcomes the combinatorial explosion in communicating structurally complex data. Internal representation can be made more general by letting part of the representation determine how rest of the assemblies should be interpreted, i.e. by using *data-specific assemblies*. These techniques are implemented in a recurrent FGREP network. A sequential input network reads a sequence of input items into

General

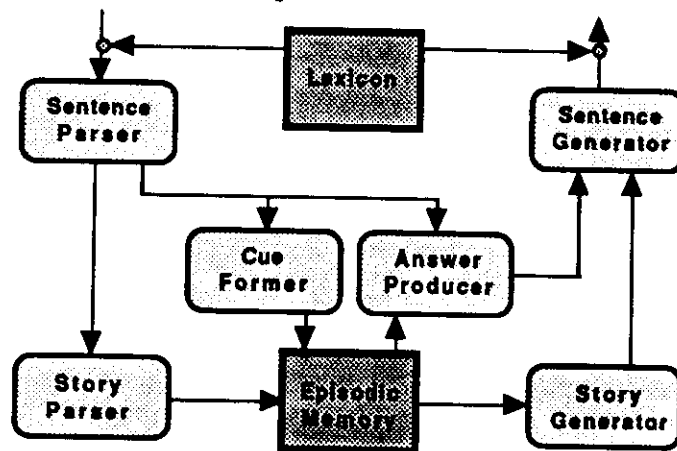
- modular architecture
- central lexicon
- dissociated training
- automatic error clean-up

Lexicon

- filter translating between lexical <-> semantic reps
- many-to-many mapping
- disambiguation and priming (future)
- dyslexic errors, aphasic impairments
- filtering noise

Representation

- orthographic I/O reps.
- holographic internal rep.
- publicly available reps.
- data-specific assemblies



EGREP modules

- reactive training env.
- developing representations
- ID+content technique
- hierarchical organization
- sequential communication
- expectations
- filtering noise

Hierarchical feature maps

- taxonomy & topology
- concentrating resources
- efficient self-organization
- filtering noise

Trace feature maps

- one shot storage
- associative retrieval
- completion, error correction
- recency, uniqueness

Figure 14.1: The main features and contributions of the DISCERN model.

a stationary output representation, displaying expectations about the full context of each item. A sequential output network produces a sequence of output items from stationary input.

The FGREP modules, together with a central lexicon, are used as the basic processing building blocks in DISCERN. The processing is carried out by a *hierarchical organization* of six FGREP modules, trained to paraphrase and answer questions about script-based stories using natural language input and output. The complexity of this task is reduced by effectively dividing it into subgoals. Each module is *trained separately and in parallel*, each developing the same lexicon. DISCERN is able to answer questions and produce a fully expanded paraphrase of a story from only a few sentences, i.e. the unmentioned events are inferred. The word instances are correctly bound to their roles, and simple plausible inferences of the variable content of the story are made in the process. The role binding errors that occur are plausible, i.e. fillers are confused with plausible alternatives, and these bindings are maintained throughout the paraphrase and question answering.

The memory modules form an integral part of DISCERN, although they have very different processing requirements. Categorical perception, parallel representation of alternatives without blending, and storing items with a single presentation are some of their unique characteristics.

The lexicon consists of the lexical and semantic memories, which are laid out on separate feature maps. The lexicon acts as a *filter*, translating each lexical input symbol representation into the internal semantic representation, and inversely translating at the output. The mapping between the symbols and concepts is *many-to-many*, and supports mechanisms for disambiguation and semantic priming. The lexicon also efficiently filters out noise in the translation process. As long as the input is recognizable, the output is exactly correct.

The lexical memory is organized according to the physical similarities of the symbols, and the semantic memory lays out the similarities in the meanings of words. This makes it possible to model *dyslexic performance errors*. Localized lesions to the lexical and semantic maps and pathways between them cause *category-specific impairments* similar to ones occurring in acquired aphasia.

The episodic memory classifies an input story as an instance of a particular script, track and role binding, and assigns a unique memory location for it. The recognition taxonomy, i.e. the breakdown of each script into the tracks and roles, is extracted automatically and independently for each script from examples of script instantiations in an unsupervised self-organizing process. The process resembles human learning in that the differentiation of the most frequently encountered scripts become gradually the most detailed.

The resulting structure is a *hierachical pyramid of feature maps*. The stories are represented in the episodic memory by their image units at the script, track and role-binding levels. The hierarchy *visualizes the taxonomy* and the maps *lay out the topology* of each level. The most salient parts of the input data are separated and most *resources are concentrated* on representing them accurately. Self-organization is very fast because it employs *hierar-chical subgoalng*. The classification performed by the episodic memory is very robust and efficiently filters out noise in the input representation.

Each script-based story has a unique location on one of the role-binding maps at the bottom of the episodic memory hierarchy. A trace is created at that location with the *trace feature map* mechanism. The lateral connections on that map are modified so that they are "pointing" to that location. Because different items are represented at different parts of the hardware, they can be stored with only a *single presentation*, without wiping out previous traces.

When an incomplete input cue is presented to the episodic memory, the hierarchy classifies it as an instance of a script and track, and makes a best guess about the role binding. The lateral connections complete the cue by bringing the activity to the center of the trace. More *recent* traces occupy larger areas on the map, making them more likely to be retrieved in case of an ambiguous cue. *Unique* memory traces will be preserved even when other parts of the memory become overloaded. It is possible to recognize when there is nothing appropriate in the memory, and it is possible to search for a memory trace in several maps in parallel, in case of a highly ambiguous cue.

14.2 Conclusion

The DISCERN project had two main goals: (1) to show that a complete NLP system can be built from distributed artificial neural networks, and (2) to show that several high-level phenomena can be explained at the physical level using the special properties of these networks.

DISCERN's existence proves that the first goal is feasible. The distributed neural network approach is quite effective in script processing. Scripts are regular event sequences, and their structure can easily be extracted by a neural network system. Script inferences are intuitive, immediate and occur without conscious control, a process which automatically arises from generalization and graceful degradation in distributed networks.

The scale-up properties of the DISCERN approach seem very good. Hierarchical modular structure with sequential communication efficiently reduces the complexity of the task. The modules filter out each others errors, and the system performance is stable. The tasks of processing and memory are separated and implemented with network architectures which best suit their requirements. With meaningful intermediate representations it is possible to add more modules to the existing system, and these additions do not slow down the learning. It is possible to bootstrap the system with a small vocabulary, quickly increase its processing capability with the ID+content technique, and gradually learn more accurate processing. Perhaps most significantly, DISCERN processes both semantic (general, statistical) and episodic (specific, one-shot) information and is able to produce sequential high-level behaviour. We see this as a promising beginning in modeling complex cognitive tasks with neural networks.

DISCERN grounds several high-level phenomena in connectionist processes. Learning of word meanings, script processing and episodic memory organization are based on backpropagation learning and self-organization. Script-based inferences, expectations and defaults automatically result from generalization and graceful degradation in distributed networks. Several types of performance errors in role binding, episodic memory and lexical access are

explained by the physical nature of the model. Certain aphasic impairments can be modeled by physical damage to the feature maps and pathways in DISCERN.

The DISCERN project has also separately addressed several central issues in cognitive PDP research. DISCERN provides a framework for studying the formation of word meanings, modularity, episodic memory, distributed lexicon, type/token processing, role binding and sequential processing. We believe the contributions to these issues will be useful in their own right, not just as constituents of DISCERN.

Bibliography

- Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147-169.
- Allman, J. M., Baker, J. F., Newcome, W. T., and Petersen, S. E. (1981). Visual topography and function: Cortical-visual areas in the owl monkey. In Woolsey, C. N., editor, *Multiple Visual Areas, Vol. 2: Cortical Memory Organization*. Humana Press.
- Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings of the IEEE First Annual International Conference on Neural Networks*. IEEE.
- Alvarado, S., Dyer, M. G., and Flowers, M. (in press). Argument comprehension and retrieval for editorial text. *Knowledge-Based Systems*.
- Anderson, J. A. and Mozer, M. C. (1981). Categorization and selective neurons. In *Parallel Models of Associative Memory*. Lawrence Erlbaum Associates.
- Anderson, J. R. (1983). *The Architecture of Cognition*. Harvard University Press, Cambridge, MA.
- Arens, Y. (1986). *CLUSTER: An Approach to Contextual Language Understanding*. PhD thesis, Computer Science Division, University of California, Berkeley.
- Ash, T. (1989). Dynamic node creation in backpropagation networks. Technical Report ICS-8901, Institute for Cognitive Science, University of California, San Diego.
- Ballard, D. H. (1986). Cortical connections and parallel processing: Structure and function. *The Behavioral and Brain Sciences*, (9):67-120.
- Ballard, D. H. (1987). Modular learning in neural networks. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Barto, A. G., Sutton, R. S., and Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40(3):201-211.
- Basso, A., Taborelli, A., and Vignolo, L. A. (1978). Dissociated disorders of speaking and writing in aphasia. *Journal of Neurology, Neurosurgery and Psychiatry*, 41:526-556.
- Bower, G. H., Black, J. B., and Turner, T. J. (1979). Scripts in memory for text. *Cognitive Psychology*, (11):177-220.

- Brousse, O. and Smolensky, P. (1989). Virtual memories and massive generalization in connectionist combinatorial learning. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Buchanan, B. G. and Shortliffe, E. H., editors (1985). *Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA.
- Caramazza, A. (1988). Some aspects of language processing revealed through the analysis of acquired aphasia: The lexical system. *Annual Reviews in Neuroscience*, 11:395-421.
- Charniak, E. (1986). A neat theory of marker passing. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Coltheart, M., Patterson, K., and Marshall, J. C., editors (1980). *Deep Dyslexia*. International Library of Psychology. Routledge and Kegan Paul.
- Cottrell, G. W. and Small, S. L. (1983). A connectionist scheme for modelling word sense disambiguation. *Cognition and Brain Theory*, 6(1):89-120.
- Cullingford, R. E. (1978). *Script Application: Computer Understanding of Newspaper Stories*. PhD thesis, Department of Computer Science, Yale University. Technical Report 116.
- Curtiss, S. (1988). The special talent of grammar acquisition. In Obler, L. K. and Fein, D., editors, *The Exceptional Brain: Neuropsychology of Talent and Special Abilities*, chapter 20. Guilford Press, New York.
- DeJong, G. F. (1979). *Skimming Stories in Real Time: An Experiment in Integrated Understanding*. PhD thesis, Department of Computer Science, Yale University. Research Report 158.
- Derthick, M. (1988). *Mundane Reasoning by Parallel Constraint Satisfaction*. PhD thesis, Computer Science Department, Carnegie Mellon University. Technical Report CMU-CS-88-182.
- Derthick, M. and Plaut, D. C. (1986). Is distributed connectionism compatible with the physical symbol system hypothesis? In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.
- Dolan, C. P. (1989). *Tensor Manipulation Networks: Connectionist and Symbolic Approaches to Comprehension, Learning and Planning*. PhD thesis, Computer Science Department, UCLA.
- Dolan, C. P. and Dyer, M. G. (1987). Symbolic schemata, role binding, and the evolution of structure in connectionist memories. In *Proceedings of the IEEE First Annual International Conference on Neural Networks*. IEEE.

- Dolan, C. P. and Smolensky, P. (1989). Implementing a connectionist production system using tensor products. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Duda, R. O., Hart, P. E., and Nilsson, N. J. (1977). Development of a computer-based consultant for mineral exploration. Technical report, Stanford Research Institute.
- Dyer, M. G. (1983). *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*. MIT Press, Cambridge, MA.
- Dyer, M. G. (1990). Symbolic NeuroEngineering for natural language processing: A multi-level research approach. In Barnden, J. and Pollack, J., editors, *Advances in Connectionist and Neural Computation Theory*. Ablex Publ.
- Dyer, M. G., Cullingford, R. E., and Alvarado, S. (1987). Scripts. In Shapiro, S. C., editor, *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York.
- Elman, J. L. (1989). Structured representations and connectionist models. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179-211.
- Fahlman, S. E. (1977). *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press.
- Fahlman, S. E. (1988). An empirical study of learning speed in backpropagation networks. Technical Report CMU-CS-88-162, Computer Science Department, Carnegie Mellon University.
- Feldman, J. A. (1989). Neural representation of conceptual knowledge. In Nadel, Culicover, and Harnish, editors, *Neural Connections, Mental Computation*. MIT Press, Cambridge, MA.
- Fellenz, C. B. (1989). A connectionist model of linguistic analysis. Master's thesis, California State University, Chico.
- Fillmore, C. J. (1968). The case for case. In Bach, E. and Harms, R. T., editors, *Universals in Linguistic Theory*. Holt, Rinehart and Winston, New York.
- Gasser, M. (1988). *A Connectionist Model of Sentence Generation in a First and Second Language*. PhD thesis, Computer Science Department, UCLA.
- Gigley, H. (1988). Process synchronization, lexical ambiguity resolution and aphasia. In Small, S. L., Cottrell, G. W., and Tanenhaus, M. K., editors, *Lexical Ambiguity Resolution*, chapter 9. Morgan Kaufmann Publishers, Inc., Los Altos, CA.

- Golden, R. M. (1986). Representing causal schemata in connectionist systems. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.
- Goodman, R. A. and Caramazza, A. (1986). Aspects of the spelling process: Evidence from a case of acquired dysgraphia. *Language and Cognitive Processes*, 1(4):263-296.
- Graesser, A. C., Gordon, S. E., and Sawyer, J. D. (1979). Recognition memory for typical and atypical items in scripted activities: Tests for the script pointer+tag hypothesis. *Journal of Verbal Learning and Verbal Behaviour*, (18):319-332.
- Granger, R. H., Eiselt, K. P., and Holbrook, J. K. (1986). Parsing with parallelism: A spreading activation model of inference processing during text understanding. In Kolodner, J. L. and Riesbeck, C. K., editors, *Experience, Memory and Reasoning*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Grossberg, S. (1987). Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11:23-63.
- Halgren, E. (personal communication). 1990.
- Harnad, S. (1989). The symbol grounding problem. In *CNLS Conference on Emergent Computation*.
- Harris, C. L. and Elman, J. L. (1989). Representing variable information with simple recurrent networks. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Hart, J., Berndt, R. S., and Caramazza, A. (1985). Category-specific naming deficit following cerebral infarction. *Nature*, 316(1):439-440.
- Hartigan, J. A. (1975). *Clustering Algorithms*. Wiley.
- Hartline, H. K. (1949). Inhibition of activity of visual receptors by illuminating nearby retinal areas in the limulus eye. *Federation Proceedings*, 8(1):69-.
- Heit, G., Smith, M. E., and Halgren, E. (1989). Neural encoding of individual words and faces by the human hippocampus and amygdala. *Nature*, (333):773-775.
- Hendler, J. (1988). *Integrating Marker Passing and Problem Solving: A Spreading Activation Approach to Improved Choice in Planning*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Hinton, G. E. (1981). Implementing semantic networks in parallel hardware. In Hinton, G. E. and Anderson, J. A., editors, *Parallel Models for Associative Memory*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.

- Hinton, G. E. (1987). Connectionist learning procedures. Technical Report CMU-CS-87-115, Computer Science Department, Carnegie Mellon University.
- Hinton, G. E., McClelland, J. L., and Rumelhart, D. E. (1986). Distributed representations. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*. MIT Press, Cambridge, MA.
- Hinton, G. E. and Plaut, D. C. (1987). Using fast weights to deblur old memories. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.
- Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences, USA*, pages 2554–2558.
- Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. In *Proceedings of the National Academy of Science, USA*, pages 3088–3092.
- Inhelder, B. and Piaget, J. (1958). *The Growth of Logical Thinking from Childhood to Adolescence*. Basic Books, New York.
- Jacobs, R. A. (1990). *Task Decomposition Through Competition in a Modular Connectionist Architecture*. PhD thesis, Department of Computer and Information Science, University of Massachusetts at Amherst.
- Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.
- Kanerva, P. (1988). *Sparse Distributed Memory*. MIT Press, Cambridge, MA.
- Kawamoto, A. H. (1988). Distributed representations of ambiguous words and their resolution in a connectionist network. In Small, S. L., Cottrell, G. W., and Tanenhaus, M. K., editors, *Lexical Ambiguity Resolution*, chapter 8. Morgan Kaufmann Publishers, Inc., Los Altos, CA.
- Kohonen, T. (1982a). Analysis of a simple self-organizing process. *Biological Cybernetics*, (44):135–140.
- Kohonen, T. (1982b). Clustering, taxonomy, and topological maps of patterns. In *Proceedings of the Sixth International Conference on Pattern Recognition*. IEEE Computer Society Press.
- Kohonen, T. (1982c). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, (43):59–69.
- Kohonen, T. (1984). *Self-Organization and Associative Memory*. Springer-Verlag, Berlin; New York.

- Kolodner, J. L. (1984). *Retrieval and Organizational Strategies in Conceptual Memory*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Korf, R. E. (1987). Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65-88.
- Kosslyn, S. M. and Hatfield, G. (1984). Representation without symbol systems. *Social Research*, 51:1019-1045.
- Lakoff, G. (1989). A suggestion for a linguistics with connectionist foundations. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Lange, T. E. (in press). Lexical and pragmatic disambiguation and reinterpretation in connectionist networks. *International Journal of Man-Machine Studies*.
- Lange, T. E. and Dyer, M. G. (1989). High-level inferencing in a connectionist network. *Connection Science*, 1(2).
- Lashley, K. S. (1950). In search of the engram. In *Society of Experimental Biology Symposium No. 4: Psychological Mechanisms in Animal Behaviour*, pages 478-505, London. Cambridge University Press.
- Lebowitz, M. (1980). *Generalization and Memory in an Integrated Understanding System*. PhD thesis, Department of Computer Science, Yale University. Research Report 186.
- Lee, G. (1990). DYNASTYII: A distributed connectionist plan applier mechanism. Unpublished Research Report.
- Lee, G., Flowers, M., and Dyer, M. G. (1989). A symbolic /connectionist script applier mechanism. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Lehnert, W. G. (1978). *The Process of Question Answering*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Loftus, E. F. (1975). Leading questions and the eyewitness report. *Cognitive Psychology*, (7):560-572.
- McCarthy, R. A. and Warrington, E. K. (1988). Evidence for modality-specific meaning systems in the brain. *Nature*, 334(4):428-430.
- McClelland, J. L. and Kawamoto, A. H. (1986). Mechanisms of sentence processing: Assigning roles to constituents. In McClelland, J. L. and Rumelhart, D. E., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume II: Psychological and Biological Models*. MIT Press, Cambridge, MA.

- McClelland, J. L., Rumelhart, D. E., and Hinton, G. E. (1986). The appeal of parallel distributed processing. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*. MIT Press, Cambridge, MA.
- Miceli, G., Silveri, M. C., and Caramazza, A. (1985). Cognitive analysis of a case of pure dysgraphia. *Brain and Language*, 25:187-212.
- Miikkulainen, R. (1987). Self-organizing process based on lateral inhibition and weight redistribution. Technical Report UCLA-AI-87-16, Artificial Intelligence Laboratory, Computer Science Department, University of California, Los Angeles.
- Miikkulainen, R. (1990a). A distributed feature map model of the lexicon. In *Proceedings of the Twelfth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.
- Miikkulainen, R. (1990b). A PDP architecture for processing sentences with relative clauses. In *Proceedings of the 13th International Conference on Computational Linguistics*.
- Miikkulainen, R. (in press). Script recognition with hierarchical feature maps. *Connection Science*.
- Miikkulainen, R. and Dyer, M. G. (1987). Building distributed representations without microfeatures. Technical Report UCLA-AI-87-17, Artificial Intelligence Laboratory, Computer Science Department, University of California, Los Angeles.
- Miikkulainen, R. and Dyer, M. G. (1989a). Encoding input/output representations in connectionist cognitive systems. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Miikkulainen, R. and Dyer, M. G. (1989b). A modular neural network architecture for sequential paraphrasing of script-based stories. In *Proceedings of the International Joint Conference on Neural Networks*. IEEE.
- Minsky, M. (1963). Steps toward artificial intelligence. In Feigenbaum, E. A. and Feldman, J. A., editors, *Computers and Thought*. McGraw-Hill, New York.
- Minsky, M. (1981). A framework for representing knowledge. In Haugeland, J., editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence*. MIT Press, Cambridge, MA.
- Minsky, M. (1985). *Society of Mind*. Simon and Schuster, New York.
- Minsky, M. and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, expanded edition.
- Mozer, M. C. (1988). A focused back-propagation algorithm for temporal pattern recognition. Technical Report CRG-TR-88-3, Connectionist Research Group, Departments of Psychology and Computer Science, University of Toronto.

- Nenov, V. I., Read, W., Halgren, E., and Dyer, M. G. (1990). The effects of threshold modulation on recall & recognition in a sparse auto-associative memory: Implications for hippocampal physiology. In *Proceedings of the International Joint Conference on Neural Networks, Washington, DC*.
- Osherson, D. N. (1974). *Logical Abilities in Children*. Lawrence Erlbaum Associates.
- Pazzani, M. J. (1988). *Learning Causal Relationships: An Integration of Empirical and Explanation-Based Learning Methods*. PhD thesis, Computer Science Department, University of California, Los Angeles. Technical Report UCLA-AI-88-10.
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., Los Altos, CA.
- Pineda, F. L. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229-2232.
- Plunkett, K. and Marchman, V. (1989). Pattern association in a back propagation network: Implications for child language acquisition. Technical Report 8902, Center for Research in Language, University of California, San Diego.
- Plutowski, M. E. P. (1988). Backpropagation with higher-order units. ICNN-88 poster.
- Pollack, J. B. (1987). Cascaded back-propagation on dynamic connectionist networks. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Pollack, J. B. (1988). Recursive auto-associative memory: Devising compositional distributed representations. In *Proceedings of the Tenth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Pollack, J. B. (1989). Implications of recursive distributed representations. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems I*. Morgan Kaufmann Publishers, Inc., Los Altos, CA.
- Quillian, M. R. (1967). Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science*, (12):410-430.
- Ratliff, F., Hartline, H. K., and Lange, D. (1966). The dynamics of lateral inhibition in the compound eye of limulus. In *Proceedings of the International Symposium on the Functional Organization of the Compound Eye*.
- Regier, T. (1989). Recognizing image-schemas using programmable networks. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Ritter, H. (1989). Combining self-organizing maps. In *Proceedings of the International Joint Conference on Neural Networks*. IEEE.

- Ritter, H. J. and Schulzen, K. J. (1988). Convergency properties of kohonen's topology conserving maps: Fluctuations, stability and dimension selection. *Biological Cybernetics*, 60:59-71.
- Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington.
- Rumelhart, D. E., Hinton, G. E., and McClelland, J. L. (1986a). A general framework for parallel distributed processing. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*. MIT Press, Cambridge, MA.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*. MIT Press, Cambridge, MA.
- Rumelhart, D. E. and McClelland, J. L. (1987). Learning the past tenses of English verbs: Implicit rules or parallel distributed processing. In MacWhinney, B., editor, *Mechanisms of Language Acquisition*. Erlbaum, Hillsdale, NJ.
- Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986c). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge, MA.
- Rumelhart, D. E., Smolensky, P., McClelland, J. L., and Hinton, G. E. (1986d). Schemata and sequential thought processes in PDP models. In McClelland, J. L. and Rumelhart, D. E., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume II: Psychological and Biological Models*. MIT Press, Cambridge, MA.
- Rumelhart, D. E. and Zipser, D. (1986). Feature discovery by competitive learning. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*. MIT Press, Cambridge, MA.
- Sanford, A. J. and Garrod, S. C. (1981). *Understanding Written Language*. Wiley, Chichester.
- Sarno, M. T., editor (1981). *Acquired Aphasia*. Academic Press, New York.
- Schank, R. (1981). Language and memory. In Norman, D. A., editor, *Perspectives on Cognitive Science*. Ablex Publishing, Norwood, N.J.
- Schank, R. (1982). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, Cambridge.
- Schank, R. and Abelson, R. (1977). *Scripts, Plans, Goals, and Understanding - An Inquiry into Human Knowledge Structures*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Schank, R. and Riesbeck, C. K., editors (1981). *Inside Computer Understanding*. Lawrence Erlbaum Associates, Hillsdale, NJ.

- Sejnowski, T. J. and Rosenberg, C. R. (1986). NETtalk: A parallel network that learns to read aloud. Technical Report JHU/EECS-86/01, Electrical Engineering and Computer Science, Johns Hopkins University.
- Sejnowski, T. J. and Rosenberg, C. R. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, (1):145-168.
- Servan-Schreiber, D., Cleeremans, A., and McClelland, J. L. (1989). Learning sequential structure in simple recurrent networks. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems I*. Morgan Kaufmann Publishers, Inc., Los Altos, CA.
- Sharkey, N. E. and Mitchell, D. C. (1985). Word recognition in a functional context: The use of scripts in reading. *Journal of Verbal Learning and Verbal Behaviour*, (24):253-270.
- Sharkey, N. E. and Sharkey, A. J. C. (1987). What is the point of integration? the loci of knowledge-based facilitation in sentence processing. *Journal of Verbal Learning and Verbal Behaviour*, (26):255-276.
- Sharkey, N. E., Sutcliffe, R. F. E., and Wobcke, W. R. (1986). Mixing binary and continuous connection schemes for knowledge access. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Shastri, L. (1988). *Semantic Networks: An Evidential Formalization and its Connectionist Realization*. Morgan Kaufmann Publishers, Inc.
- Simon, H. (1981). *The Sciences of the Artificial*. MIT Press, Cambridge, MA.
- Simpson, G. B. and Burgess, C. (1985). Activation and selection processes in the recognition of ambiguous words. *Journal of Experimental Psychology: Human Perception and Performance*, 11(1):28-39.
- Small, S. L. (1990). Learning lexical knowledge in context: Experiments with recurrent feed forward networks. In *Proceedings of the Twelfth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.
- Smolensky, P. (1987). A method for connectionist variable binding. Technical Report CU-CS-356-87, Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder.
- St. John, M. F. (1990). *The Story Gestalt - Text Comprehension by Cue-Based Constraint Satisfaction*. PhD thesis, Department of Psychology, Carnegie Mellon University.
- St. John, M. F. and McClelland, J. L. (1989). Applying contextual constraints in sentence comprehension. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- St. John, M. F. and McClelland, J. L. (in press). Learning and applying contextual constraints in sentence comprehension. *Artificial Intelligence*.

- Stolcke, A. (1990). Learning feature-based semantics with simple recurrent networks. Technical Report TR-90-015, International Computer Science Institute.
- Sumida, R. A. and Dyer, M. G. (1989). Storing and generalizing multiple instances while maintaining knowledge-level parallelism. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.
- Swinney, D. A. (1979). Lexical access during sentence comprehension: (Re)consideration of context effects. *Journal of Verbal Learning and Verbal Behavior*, 18:645-659.
- Touretzky, D. S. (1986). Boltzcons: Reconciling connectionism with recursive structure of stacks and trees. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.
- Touretzky, D. S. (1987). Representing conceptual structures in a neural network. In *Proceedings of the IEEE First Annual International Conference on Neural Networks*. IEEE.
- Touretzky, D. S. (1989a). Connectionism and compositional semantics. Technical Report CMU-CS-89-147, Computer Science Department, Carnegie Mellon University.
- Touretzky, D. S. (1989b). Towards a connectionist phonology: The "many maps" approach to sequence manipulation. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*.
- Touretzky, D. S. and Hinton, G. E. (1986). A distributed connectionist production system. Technical Report CMU-CS-86-172, Computer Science Department, Carnegie Mellon University.
- van Gelder, T. (1989). *Distributed Representation*. PhD thesis, Department of Philosophy, University of Pittsburgh.
- Waibel, A., Sawai, H., and Shikano, K. (1988). Modularity and scaling in large phonemic neural networks. Technical Report TR-I-0034, Advanced Telecommunication Research Institute, International Interpreting Telephony Research Laboratories.
- Waltz, D. L. and Pollack, J. B. (1985). Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9:51-74.
- Warrington, E. K. (1975). The selective impairment of semantic memory. *Quarterly Journal of Experimental Psychology*, 27:635-657.
- Warrington, E. K. and McCarthy, R. A. (1987). Categories of knowledge: Further fractionations and an attempted integration. *Brain*, 110:1273-1296.
- Warrington, E. K. and Shallice, T. (1984). Category specific semantic impairments. *Brain*, 107:829-854.
- Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270-280.

Zernik, U. (1987). *Strategies of Language Acquisition: Learning Phrases from Examples in Context*. PhD thesis, Computer Science Department, University of California, Los Angeles. Technical Report UCLA-AI-87-1.

APPENDIX A

Story data

The story skeletons and questions for each track are listed below. The slots (i.e. roles) are the same for all tracks within a script, while the role bindings vary from track to track. The words in upper case are prototype words. The actual stories are formed from these skeletons by specifying the ID part for each prototype word that appears in the skeleton. The incomplete stories for testing purposes were formed from sentences marked with "*".

The sentences were represented with six case-role assemblies: agent, act, recipient, patient-attribute, patient and location. This means that certain related case-roles, e.g. from-, to- and at-location were actually represented by the same assembly for simplicity. The interpretation of this assembly depends on the value in the act-assembly, i.e. the representation is data-specific.

RESTAURANT SCRIPT

Slots: script, track, customer, food, restaurant, taste, tip

Fancy-restaurant track

Fillers: \$restaurant, \$fancy, PERSON, FANCY-FOOD, FANCY-REST, good, big/small

Instance names: John, Mary, lobster, steak, MaMaison, Leone's

PERSON went to FANCY-REST.*
The waiter seated PERSON.
PERSON asked the waiter for FANCY-FOOD.*
PERSON ate a good FANCY-FOOD.
PERSON paid the waiter.
PERSON left a big/small tip.*
PERSON left FANCY-REST.

Who ate FANCY-FOOD at FANCY-REST?
PERSON did.
What did PERSON eat at FANCY-REST?
PERSON ate a good FANCY-FOOD.
Where did PERSON eat FANCY-FOOD?
PERSON ate FANCY-FOOD at FANCY-REST.
How did PERSON like FANCY-FOOD at FANCY-REST?
PERSON thought FANCY-FOOD was good at FANCY-REST.

Coffee-shop-restaurant track

Fillers: \$restaurant, \$coffee, PERSON, COFFEE-FOOD, COFFEE-REST, good/bad, big/small

Instance names: John, Mary, spaghetti, fish, Denny's, Norms

PERSON went to COFFEE-REST.*
PERSON seated PERSON.
PERSON asked the waiter for COFFEE-FOOD.*
PERSON ate a good/bad COFFEE-FOOD.*
PERSON left a big/small tip.
PERSON paid the cashier.
PERSON left COFFEE-REST.

Who ate COFFEE-FOOD at COFFEE-REST?
PERSON did.
What did PERSON eat at COFFEE-REST?
PERSON ate a good/bad COFFEE-FOOD.
Where did PERSON eat COFFEE-FOOD?
PERSON ate COFFEE-FOOD at COFFEE-REST.
How did PERSON like COFFEE-FOOD at COFFEE-REST?
PERSON thought COFFEE-FOOD was good/bad at COFFEE-REST.

Fast-food-restaurant track

Fillers: \$restaurant, \$fast, PERSON, FAST-FOOD, FAST-REST, bad, none
Instance names: John, Mary, hamburger, fries, McDonald's, BurgerKing

PERSON went to FAST-REST.*
PERSON asked the cashier for FAST-FOOD.*
PERSON paid the cashier.
PERSON seated PERSON.
PERSON ate the small FAST-FOOD.*
The FAST-FOOD tasted bad.
PERSON left FAST-REST.

Who ate FAST-FOOD at FAST-REST?
PERSON did.
What did PERSON eat at FAST-REST?
PERSON ate a small FAST-FOOD.
Where did PERSON eat FAST-FOOD?
PERSON ate FAST-FOOD at FAST-REST.
How did PERSON like FAST-FOOD at FAST-REST?
PERSON thought FAST-FOOD was bad at FAST-REST.

SHOPPING SCRIPT

Slots: script, track, customer, item, store

Clothing-shopping track

Fillers: \$shopping, \$clothing, PERSON, CLOTH-ITEM, CLOTH-STORE
Instance names: John, Mary, shoes, hat, Broadway, Bullock's

PERSON went to CLOTH-STORE.*
PERSON looked for good CLOTH-ITEM.*
PERSON tried on several CLOTH-ITEM.
PERSON took the best CLOTH-ITEM.*

PERSON paid the cashier.
PERSON left CLOTH-STORE.

Who bought CLOTH-ITEM at CLOTH-STORE?
PERSON did.

What did PERSON buy at CLOTH-STORE?

PERSON bought CLOTH-ITEM.

Where did PERSON buy CLOTH-ITEM?

PERSON bought CLOTH-ITEM at CLOTH-STORE.

Electronics-shopping track

Fillers: \$shopping, \$electronics, PERSON, ELECTR-ITEM, ELECTR-STORE
Instance names: John, Mary, CD-player, TV, RadioShack, CircuitCty

PERSON went to ELECTR-STORE.*

PERSON looked for good ELECTR-ITEM.*

PERSON asked the staff questions about ELECTR-ITEM.

PERSON took the best ELECTR-ITEM.*

PERSON paid the cashier.

PERSON left ELECTR-STORE.

Who bought ELECTR-ITEM at ELECTR-STORE?

PERSON did.

What did PERSON buy at ELECTR-STORE?

PERSON bought ELECTR-ITEM.

Where did PERSON buy ELECTR-ITEM?

PERSON bought ELECTR-ITEM at ELECTR-STORE.

Grocery-shopping track

Fillers: \$shopping, \$grocery, PERSON, GROCERY-ITEM, GROCERY-STORE
Instance names: John, Mary, softdrinks, vegetables, Ralph's, Safeway

PERSON went to GROCERY-STORE.*

PERSON took a big shopping-cart.

PERSON compared GROCERY-ITEM prices.*

PERSON took several GROCERY-ITEM.*

PERSON waited in a big line.

PERSON paid the cashier.

PERSON left GROCERY-STORE.

Who bought GROCERY-ITEM at GROCERY-STORE?

PERSON did.

What did PERSON buy at GROCERY-STORE?

PERSON bought GROCERY-ITEM.

Where did PERSON buy GROCERY-ITEM?

PERSON bought GROCERY-ITEM at GROCERY-STORE.

TRAVEL SCRIPT

Slots: script, track, traveler, origin, destination, distance

Plane-travel track

Fillers: \$travel, \$plane, PERSON, PLANE-ORIGIN, PLANE-DEST, big
Instance names: John, Mary, LAX, DFW, JFK, SFO

PERSON went to PLANE-ORIGIN.*
PERSON checked-in for a flight to PLANE-DEST.*
PERSON waited at the gate for boarding.
PERSON got-on the plane.
The plane took-off from PLANE-ORIGIN.
The plane arrived at PLANE-DEST.*
PERSON got-off the plane.

Who traveled to PLANE-DEST?
PERSON did.
What did PERSON take to PLANE-DEST?
PERSON took a plane.
Where did PERSON take a plane to?
PERSON took a plane to PLANE-DEST.
Did PERSON travel big distance to PLANE-DEST?
PERSON traveled a big distance.

Train-travel track

Fillers: \$travel, \$train, PERSON, TRAIN-ORIGIN, TRAIN-DEST, big/small
Instance names: John, Mary, CentralSta, DowntownSt, Boston, NewYork

PERSON went to TRAIN-ORIGIN.*
PERSON bought a ticket to TRAIN-DEST.*
PERSON got-on the train.
The conductor punched the ticket.
PERSON traveled a big/small distance.
PERSON got-off at TRAIN-DEST.*

Who traveled to TRAIN-DEST?
PERSON did.
What did PERSON take to TRAIN-DEST?
PERSON took a train.
Where did PERSON take a train to?
PERSON took a train to TRAIN-DEST.
Did PERSON travel big distance to TRAIN-DEST?
PERSON traveled a big/small distance.

Bus-travel track

Fillers: \$travel, \$bus, PERSON, BUS-ORIGIN, BUS-DEST, small
Instance names: John, Mary, bus-stop, bus-termin, downtown, beach

PERSON went to BUS-ORIGIN.*
PERSON waited for the bus.*
PERSON got-on the BUS-DEST bus.
PERSON paid the driver.
The bus arrived at BUS-DEST.*
PERSON got-off the bus.

Who traveled to BUS-DEST?

PERSON did.

What did PERSON take to BUS-DEST?

PERSON took a bus.

Where did PERSON take a bus to?

PERSON took a bus to BUS-DEST.

Did PERSON travel big distance to BUS-DEST?

PERSON traveled a small distance.

APPENDIX B

Implementation details

The code for DISCERN and the various test routines were developed on an HP 9000/350 workstation with 16M memory, using standard C, the HP Starbase graphics package, and Lucid Common Lisp. The FGREP modules were trained on Cray Y-MP 8/864 at San Diego Supercomputer Center. The lexicon, hierarchical feature maps and single-level feature maps were developed on an IBM 3090/600S, running AIX, at the Office of Academic Computing at UCLA. The code (other than graphics) is completely portable between these systems.

Approximately 150 training, testing and analysis programs were developed in the DISCERN project, a total of about 130,000 lines of C code and 15,000 lines of Common Lisp. Not all the code is unique, of course. Most of the programs implement different versions of the same basic techniques, sharing similar subroutines. Training programs constitute approximately 20% of the code, 40% consists of performance tests and 40% of comparison, demonstration and analysis programs. As the tip of the iceberg, the actual DISCERN performance program (appendix D) contains 1,500 lines and the DISCERN training programs (appendix C) together about 4,000 lines.

In training and testing, graphics routines typically constitute 35% of the code, simulation management and I/O approximately another 35%, and 30% could be considered core neural network code. In the following appendices, selected portions of the code and data are listed.

APPENDIX C

DISCERN training code and data

The key parts of code for training the different modules of DISCERN are presented in this appendix. The purpose is to show concrete implementations of the training mechanisms, not to provide complete programs that could be downloaded and run. The graphics and most low-level routines have been omitted.

There are separate training programs for the FGREP modules, the lexicon and the hierarchical feature maps. The FGREP modules are trained first, and the final word representations are used to organize the lexicon and the hierarchical feature maps of the episodic memory.

Each module is trained with compatible data, based on the same set of training stories. The lisp program that generates the stories is listed in section C.1.2.

C.1 FGREP modules

C.1.1 Training code

Below is the crucial parts of the code for training the FGREP modules in DISCERN (chapter 5). Graphics code and most of the initialization procedures and file I/O have been omitted. Depending on how the code is conditionalized, all networks can be trained simultaneously on the same machine or they can be trained on separate machines. In the first case, the representations are kept in the memory, and they are modified immediately after each presentation and each network is trained equal number of epochs. In the latter case, the representations are stored in a file, they are modified only after the whole epoch, and each network is trained equally long time.

```
#include <stdio.h>
#include <math.h>

/* conditionalization */
#define reps_in_file 0      /* reps kept in file / memory */
#define batch_update 0    /* word reps updated batch/online */
#define time_params 0     /* phaseends, snapshots in secs/epochs*/
#if time_params
#define counter time(0)   /* train all nets equally long time */
#else
#define counter epoch    /* train all nets equally many epochs*/
#endif

/* data constants etc. */
#define hidrepstep 5      /* steps for optimizing inner fwd prop loops:*/
#define inpunitstep 4    /* must divide number of hidden/input units */
#define toks 1           /* tokens / no */
#define ntoks 2          /* number of id units */
#define firsttypeword 12 /* index of first type word */
```

```

#define ntypewords 19          /* number of type words */
#define batch_rep_eta_ratio 0.1 /* in batch mode, eta must be smaller*/
#define snapshotend 99999999 /* token that indicates last snapshot in file*/

/* max table dimensions */
#define maxnets 6            /* maximum number of networks */
#define maxrep 12             /* maximum size of the word representations */
#define maxhidrep 100        /* maximum size of hidden layers */
#define maxstories 18        /* maximum number of stories */
#define maxwordl 20          /* maximum length of word labels */
#define maxas 13             /* max of nword, ncase+nslot, nsent+nquest */
#define maxsnaps 50          /* maximum number of snapshots */
#define maxphase 7           /* maximum number of training phases */
#define maxinputs (maxstories*maxas)
#define maxunits (maxas*maxrep) /* largest possible layer */
#define maxwords 200         /* maximum number of words */

/* simulation parameters */
int fildes, displaying, continuing, testing, nextsnapshot,
    nphase, startepoch, epoch, running[maxnets],
    seed, cmapsize;
int shuffletable[maxinps]; /* table of indices in this epoch */
long phaseends[maxphase], snapshots[maxsnaps];
float colors[256][3], etas[maxphase], eta, batchrepeta;

/* actual table dimensions */
int ninpas[maxnets], noutas[maxnets], ninpitem[maxnets], noutitem[maxnets],
    ninprep[maxnets], noutrep[maxnets], nhidrep[maxnets], nwords, nwordrep,
    ninpunits[maxnets], noutunits[maxnets],
    nstories, nnets, nquest;

/* representation indices */
int curinpnms[maxnets][maxas], curtchnms[maxnets][maxas],
    allinpnms[maxnets][maxinputs][maxas],
    alltchnms[maxnets][maxinputs][maxas];

/* units and weights */
float inprep[maxnets][maxunits], outrep[maxnets][maxunits],
    tchrep[maxnets][maxunits],
    hidbias[maxnets][maxhidrep], outbias[maxnets][maxunits],
    prevhidrep[maxnets][maxhidrep], hidrep[maxnets][maxhidrep],
    wih[maxnets][maxunits][maxhidrep],
    who[maxnets][maxunits][maxhidrep],
    wph[maxnets][maxhidrep][maxhidrep];

/* file names */
char simufile[100], wordfile[100], inputfile[100], repfile[100];
FILE *fp, *repfp;

/* lexicon */
float blankrep[maxrep]; /* blankrep=words[-1].rep */
struct lexicon {
    char chars[maxwordl];
    float grad[maxrep];
    float rep[maxrep];
} words[maxwords];

/***** main control *****/
main(argc,argv)
int argc; char *argv[];
{
    init_params(argv); /* (not listed) read the simulation specs */
    read_inputs(); /* (not listed) read the training corpus */
    if (continuing)
        iterate_snapshots();
}

```

```

    else init_weights();          /* (not listed) random initial weights */
    training();
    exit(0);
}

iterate_snapshots()
/* go through the saved snapshots, (displaying them and) updating rand */
{
    register int i, ii,j,k, neti;
    int oldepoche, oldtesting, readfun(), randfun();
    float cume[marnets];

    fp=fopen(simufile,"r");
    read_params(fp);              /* we have to read in the beg. of file again */
    oldtesting=testing;
    testing=1;                    /* we don't want to change weights */
    oldepoche=0;
    iterate_weights(randfun, (-1.0), 2.0, 0.0, 1.0); /* just to update rand */
    while (fscanf(fp,"%d",&epoch)!=EOF) /* read the epoch */
    {
        /* update the shuffling and parameters */
        epoch++;
        startepoch=epoch;
        for(i=0; i<epoch-oldepoche; i++)
        {
            #if toks
                for(ii=0; ii<nstories; ii++)
                    for(j=0; j<ntypewords; j++)
                        for(k=0; k<ntoks; k++)
                            rand();
            #endif
                shuffle();
        }
        oldepoche=epoch;

        /* read the current word representations and weights */
        iterate_weights(readfun);
    }
    fclose(fp);
    /* get the next snapshot after where we are now */
    for (nextsnapshot=0; counter > snapshots[nextsnapshot]; nextsnapshot++){
    #if reps_in_file
        save_reps();              /* make sure we continue from current reps */
    #endif
        testing=oldtesting;
    }

training()
{
    for(epoch=startepoch; counter<=phaseends[nphase-1]; epoch++)
    {
        get_current_params();
    #if batch_update
        init_batch();            /* (not listed) gradients = 0 */
    #endif
        /* one epoch = one cycle through the training corpus */
        iterate_inputs();
    #if batch_update
        save_batch();            /* (not listed) modify reps according to gradients */
    #endif
        shuffle();                /*change the order of presentations (in shuffletable)*/
        if (counter >= snapshots[nextsnapshot])
            save_current();
    }
}

```

```

iterate_inputs()
{
    int i;
    for(i=0; i<nstories; i++)
        iterate_story(shuffletable[i]);
}

iterate_story(story)
/* process one input story */
int story;
{
    int itemi, questi;
    #if toks
    randomize_tokens(); /* set up the IDs randomly for the story */
    #endif
    /* only train the specified networks (if trained on separate machines) */
    /* first train to parse and paraphrase the story */
    if(running[1])
        sequence_in(1,story); /* story parser */
    else if(running[0]) /* sentence parser */
        for(itemi=0; itemi<ninpas[1]; itemi++)
            if(allinpnms[1][story][itemi]>-1)
                sequence_in(0, story*(ninpas[1]+nquest)+itemi);
    if(running[2])
        sequence_out(2,story); /* story generator */
    else if(running[3]) /* sentence generator */
        for(itemi=0; itemi<noutas[2]; itemi++)
            if(alltchnms[2][story][itemi]>-1)
                sequence_out(3,story*(ninpas[1]+nquest)+itemi);
    /* then train to parse and answer questions */
    for(questi=0; questi<nquest; questi++)
    {
        if(allinpnms[0][story*(ninpas[1]+nquest)+ninpas[1]+questi][0]>-1)
            if(running[4]) /* cue former */
                nonrecurrent(4,story,questi);
            else if(running[0]) /* sentence parser */
                sequence_in(0, story*(ninpas[1]+nquest)+ninpas[1]+questi);
        if(alltchnms[3][story*(ninpas[1]+nquest)+ninpas[1]+questi][0]>-1)
            if(running[5]) /* answer producer */
                nonrecurrent(5,story,questi);
            else if(running[3]) /* sentence generator */
                sequence_out(3, story*(ninpas[1]+nquest)+ninpas[1]+questi);
    }
}

/***** FGREP modules *****/

sequence_in(neti,seqi)
register int neti,seqi;
{
    register int i,j,itemi;
    int *indataptr;

    /* (not listed) obtain the indices for the current input */
    init_sequence(neti,seqi);
    /* form the teaching pattern from the lexicon representations */
    for(i=0; i<noutas[neti]; i++)
        for(j=0; j<noutrep[neti]; j++)
            tchrep[neti][i*noutrep[neti]+j]=words[curtchnms[neti][i]].rep[j];

    /* process the sequence one step at a time */
    for(itemi=0; itemi<ninpas[neti]; itemi++)
        if(curinpnms[neti][itemi]>-1) /* if included */
        {
            if (neti==1 && running[0]) /* first train the sentence parser */
                sequence_in(0, seqi*(ninpas[1]+nquest)+itemi);
            if (neti==1) dataptr = alltchnms[0][curinpnms[neti][itemi]];
        }
}

```

```

        else inpdataptr = #curinpnms[neti][itemi];
        /* form the input pattern from the lexicon representations */
        for(i=0; i<ninpitem[neti]; i++)
            for(j=0; j<ninprep[neti]; j++)
                inprep[neti][i*noutrep[neti]+j]=words[indataptr[i]].rep[j];
        propagate_and_display(neti,inpdataptr);
    }
}

sequence_out(neti,seqi)
register int neti,seqi;
{
    register int i,j,itemi;
    int *tchdataptr;

    /* (not listed) obtain the indices for the current input */
    init_sequence(neti,seqi);
    /* form the input pattern from the lexicon representations */
    for (i=0; i<ninpas[neti]; i++)
        for(j=0; j<ninprep[neti];j++)
            inprep[neti][i*ninprep[neti]+j]=words[curinpnms[neti][i]].rep[j];

    /* process the sequence one step at a time */
    for(itemi=0; itemi<noutas[neti]; itemi++)
        if(curtnums[neti][itemi]>-1) /* if included */
            {
                if (neti==2) tchdataptr = alltchnms[0][curtnums[neti][itemi]];
                else tchdataptr = #curtchnms[neti][itemi];
                /* form the teaching pattern from the lexicon representations */
                for (i=0; i<noutitem[neti]; i++)
                    for(j=0; j<noutrep[neti]; j++)
                        tchrep[neti][i*noutrep[neti]+j]= words[tchdataptr[i]].rep[j];
                propagate_and_display(neti,curinpnms[neti]);
                if (neti==2 && running[3]) /* train the sentence generator also */
                    sequence_out(3,seqi*(ninpas[1]+nquest)+itemi);
            }
}

nonrecurrent(neti,seqi,questi)
register int neti,seqi,questi;
{
    register int i,j;

    /* (not listed) obtain the indices for the current input */
    init_sequence(neti,seqi*nquest+questi);
    /* form the teaching pattern from the lexicon representations */
    for(i=0; i<noutas[neti]; i++)
        for(j=0; j<noutrep[neti]; j++)
            tchrep[neti][i*noutrep[neti]+j]=words[curtchnms[neti][i]].rep[j];

    if (neti==4 && running[0]) /* first train the sentence parser */
        sequence_in(0, seqi*(ninpas[1]+nquest)+ninpas[1]+questi);
    /* form the input pattern from the lexicon representations */
    for (i=0; i<ninpas[neti]; i++)
        for(j=0; j<ninprep[neti];j++)
            inprep[neti][i*ninprep[neti]+j]=words[curinpnms[neti][i]].rep[j];
    propagate_and_display(neti,curinpnms[neti]);
    if(neti==5 && running[3]) /* train the sentence generator also */
        sequence_out(3, seqi*(ninpas[1]+nquest)+ninpas[1]+questi);
}

/***** propagation *****/
propagate_and_display(neti,inpdataptr)

```

```

/* calls to display routines are not listed, sorry */
int neti,*inptr;
{
  register int i,k;
  forward_prop(neti);
  if (!testing) backward_prop(neti,inptr);
  if (neti<4)
    for(k=0; k<nhidrep[neti]; k++) /* recurrent FGREP modules */
      prevhidrep[neti][k]=hidrep[neti][k];
}

forward_prop(neti)
/* forward propagation in the FGREP modules;
   inner loops have been rolled out for more efficient computation */
register int neti;
{
  register int i,j,k,p;

  /* set up bias for the hidden units */
  for(k=0; k<nhidrep[neti]; k++)
    hidrep[neti][k]=hidbias[neti][k];
  /* propagate from the previous hidden layer to h.l. (recurrent FGREP only) */
  if (neti<4)
    for(p=hidrepstep-1; p<nhidrep[neti]; p+=hidrepstep)
      for(k=0; k<nhidrep[neti]; k++)
        hidrep[neti][k] += prevhidrep[neti][p]*wph[neti][p][k]
          + prevhidrep[neti][p-1]*wph[neti][p-1][k]
          + prevhidrep[neti][p-2]*wph[neti][p-2][k]
          + prevhidrep[neti][p-3]*wph[neti][p-3][k]
          + prevhidrep[neti][p-4]*wph[neti][p-4][k];
  /* propagate from the input layer to the hidden layer */
  for(i=inpunitstep-1; i<ninpunits[neti]; i+=inpunitstep)
    for(k=0; k<nhidrep[neti]; k++)
      hidrep[neti][k] += inprep[neti][i]*wih[neti][i][k]
        + inprep[neti][i-1]*wih[neti][i-1][k]
        + inprep[neti][i-2]*wih[neti][i-2][k]
        + inprep[neti][i-3]*wih[neti][i-3][k];
  /* sigmoid the output layer */
  for(k=0; k<nhidrep[neti]; k++)
    hidrep[neti][k]= 1.0/(1.0+exp(-hidrep[neti][k]));

  /* set up bias for the output units */
  for(i=0; i<noutunits[neti]; i++)
    outrep[neti][i]=outbias[neti][i];
  /* propagate from the hidden layer to the output layer */
  for(k=hidrepstep-1; k<nhidrep[neti]; k+=hidrepstep)
    for(i=0; i<noutunits[neti]; i++)
      outrep[neti][i] += hidrep[neti][k]*who[neti][i][k]
        + hidrep[neti][k-1]*who[neti][i][k-1]
        + hidrep[neti][k-2]*who[neti][i][k-2]
        + hidrep[neti][k-3]*who[neti][i][k-3]
        + hidrep[neti][k-4]*who[neti][i][k-4];
  /* sigmoid the output layer */
  for(i=0; i<noutunits[neti]; i++)
    outrep[neti][i] = 1.0/(1.0+exp(-outrep[neti][i]));
}

backward_prop(neti,inptr)
int neti,*inptr;
{
  register int i,j,k,p;
  float clip(), hidsig[maxhidrep], hidfact[maxhidrep],
    outsig[maxunits], outfact[maxunits],
    hidsumsig[maxhidrep], inpsumsig[maxunits];

```

```

/* clear the error signals */
for(k=0; k<nhidrep[neti]; k++)
    hidsumsig[k]=0.0;
for(i=0; i<ninpunits[neti]; i++)
    inpsumsig[i]=0.0;

for(i=0; i<noutunits[neti]; i++)
{
    /* error signal at the output layer; equation 4.3 */
    outsig[i] = (tchrep[neti][i]-outrep[neti][i])*
        outrep[neti][i]*(1.0-outrep[neti][i]);
    outfact[i]=eta*outsig[i];
    /* modify bias units at the hidden layer
       (weight from a unit with activity always = 1.0);
       equation 4.6 */
    outbias[neti][i] += outfact[i];
}
for(i=0; i<noutunits[neti]; i++)
for(k=0; k<nhidrep[neti]; k++)
{
    /* cumulate error signal at the hidden layer; equation 4.5 */
    hidsumsig[k] += outsig[i]*who[neti][i][k];
    /* modify output weights; equation 4.6 */
    who[neti][i][k] += outfact[i]*hidrep[neti][k];
}

for(k=0; k<nhidrep[neti]; k++)
{
    /* error signal at the hidden layer; equation 4.5 */
    hidsig[k] = hidsumsig[k]*hidrep[neti][k]*(1.0-hidrep[neti][k]);
    hidfact[k] = eta*hidsig[k];
    /* modify bias units at the hidden layer
       (weight from a unit with activity always = 1.0);
       equation 4.6 */
    hidbias[neti][k] += hidfact[k];
}
for(i=0; i<ninpunits[neti]; i++)
for(k=0; k<nhidrep[neti]; k++)
{
    /* cumulate error signal at the input layer; equation 4.10 */
    inpsumsig[i] += hidsig[k]*wih[neti][i][k];
    /* modify input weights; equation 4.6 */
    wih[neti][i][k] += hidfact[k]*inprep[neti][i];
}
if (neti<4) /* recurrent fgrep module */
for(k=0; k<nhidrep[neti]; k++)
for(p=0; p<nhidrep[neti]; p++)
    /* modify previous hidden to hidden layer weights; equation 4.6 */
    wph[neti][p][k] += hidfact[k]*prevhidrep[neti][p];

/* modify representations; equations 4.11 and 4.12 */
for(i=0; i<ninpitem[neti]; i++)
if (inmdataptr[i]>-1)
{
    k=i*ninprep[neti];
    for(j=0; j<ninprep[neti]; j++)
#if batch_update
        words[inmdataptr[i]].grad[j] += inpsumsig[k+j]; /*cumulate gradient*/
#else
        words[inmdataptr[i]].rep[j] = inprep[neti][k+j] =
            clip(inprep[neti][k+j] +eta*inpsumsig[k+j]);
#endif
}
}

float clip(activity)

```

```

/* cut activity within 0 and 1; equation 4.12 */
float activity;
{
  if (activity<0.0) return(0.0);
  else if (activity > 1.0) return(1.0);
  else return(activity);
}

/***** simulation setup *****/

#if toks
randomize_tokens()
{
  register int i,j;

  for(j=0; j<ntoks; j++)
    /* the type words form a continuous block in the lexicon */
    for(i=firsttypeword; i<firsttypeword+ntypewords; i++)
      randfun(&words[i].rep[j], 0.0, 1.0);
}
#endif

```

C.1.2 Generating training data

The set of stories used in training the FGREP modules is generated by the LISP program below. There are two stories from each track in the training set (18 total), so that each track is learned equally well. Fancy-restaurant, coffee-shop-restaurant, and train-travel have two different stories (depending on the tip and the distance), other tracks have simply two copies of the same story in the training set.

Samples of the stories generated by this program are given in section C.1.3.

```

;;; generate 2 examples for each track
(defun gener ()
  (generfancy 'big)
  (generfancy 'small)
  (genercoffee 'good 'big)
  (genercoffee 'bad 'small)
  (generfast)
  (generfast)
  (generclothing)
  (generclothing)
  (generelectronics)
  (generelectronics)
  (genergrocery)
  (genergrocery)
  (generplane)
  (generplane)
  (genertrain 'big)
  (genertrain 'small)
  (generbus)
  (generbus)
)

;;; the generx functions below generate all combinations of
;;; possible role bindings

;;; generate restaurant stories
(defun generfancy (sum)
  (dolist (person '(PERSON))

```



```

(dolist (food '(FANCY-FOOD))
  (dolist (restaurant '(FANCY-REST))
    (print-out (fancy-list person food restaurant sum))))))

(defun genercoffee (taste sum)
  (dolist (person '(PERSON))
    (dolist (food '(COFFEE-FOOD))
      (dolist (restaurant '(COFFEE-REST))
        (print-out (coffee-list person food restaurant taste sum))))))

(defun generfast ()
  (dolist (person '(PERSON))
    (dolist (food '(FAST-FOOD))
      (dolist (restaurant '(FAST-REST))
        (print-out (fast-list person food restaurant))))))

;;; generate shopping stories
(defun generclothing ()
  (dolist (person '(PERSON))
    (dolist (item '(CLOTH-ITEM))
      (dolist (store '(CLOTH-STORE))
        (print-out (clothing-list person item store))))))

(defun generelectronics ()
  (dolist (person '(PERSON))
    (dolist (item '(ELECTR-ITEM))
      (dolist (store '(ELECTR-STORE))
        (print-out (electronics-list person item store))))))

(defun genergrocery ()
  (dolist (person '(PERSON))
    (dolist (item '(GROCERY-ITEM))
      (dolist (store '(GROCERY-STORE))
        (print-out (grocery-list person item store))))))

;;; generate travel stories
(defun generplane ()
  (dolist (person '(PERSON))
    (dolist (origin '(PLANE-ORIGIN))
      (dolist (dest '(PLANE-DEST))
        (print-out (plane-list person origin dest))))))

(defun genertrain (distance)
  (dolist (person '(PERSON))
    (dolist (origin '(TRAIN-ORIGIN))
      (dolist (dest '(TRAIN-DEST))
        (print-out (train-list person origin dest distance))))))

(defun generbus ()
  (dolist (person '(PERSON))
    (dolist (origin '(BUS-ORIGIN))
      (dolist (dest '(BUS-DEST))
        (print-out (bus-list person origin dest))))))

;;; format and print the story data
(defun print-out (story)
  (format t "~&")
  (dolist (row story)
    (cond ((not (null row))
           (format t "~&")
           (if (not (null (caddr row)))
               (if (equal (caar row) '_)

```

```

                (format t "-1 ")
                (format t "1 "))
        (dolist (item (append (car row) (cadr row)))
          (format t "~S " item)) ;; generate words as output
    ;; (format t "~S " (cdr (assoc item indices)))) ;; generate indices
        (if (not (null (caddr row)))
            (format t "~S" (caddr row))))))

```

```

;;; the actual input files to DISCERN consist of indices instead of words:
;;; this list is used to convert words to indices.

```

```

(setq indices
 '(
  ($restaurant . -1) ;;; blank word
  ($fancy      . 0)
  ($coffee    . 1)
  ($travel     . 2)
  ;;;etc ...
  (travel     . 99)))

```

```

;;; the generators below produce a single story with given role bindings
;;; one example of each script is given below

```

```

(defun fancy-list (person food restaurant sum)
  (list
    ;;; first print out the slot-filler rep of the story
    (list (list '$restaurant '$fancy person food restaurant 'good sum))
    ;;; then the word sequence and the case-role assignment of each sentence
    (list (list person 'went 'to restaurant '_ '_ '_ '\.)
          (list person 'went '_ '_ '_ restaurant)
          'entering)
    (list (list 'the 'waiter 'seated person '_ '_ '_ '\.)
          (list 'waiter 'seated '_ '_ person '_)
          'seating)
    (list (list person 'asked 'the 'waiter 'for food '_ '\.)
          (list person 'asked 'waiter '_ food '_)
          'ordering)
    (list (list person 'ate 'a 'good food '_ '_ '\.)
          (list person 'ate '_ 'good food '_)
          'eating)
    (list (list person 'paid 'the 'waiter '_ '_ '_ '\.)
          (list person 'paid 'waiter '_ '_ '_)
          'paying)
    (list (list person 'left 'a sum 'tip '_ '_ '\.)
          (list person 'left 'waiter sum 'tip '_)
          'tipping)
    (list (list person 'left restaurant '_ '_ '_ '\.)
          (list person 'left '_ '_ '_ restaurant)
          'leaving)
    ;;; and finally the words and case-roles for questions and answers
    (list (list 'who 'ate food 'at restaurant '_ '_ '?)
          (list 'who 'ate '_ '_ food restaurant)
          'who-question)
    (list (list person 'did '_ '_ '_ '_ '\.)
          (list person 'did '_ '_ '_ '_)
          'who-answer)
    (list (list 'what 'did person 'eat 'at restaurant '_ '?)
          (list person 'ate '_ '_ 'what restaurant)
          'what-question)
    (list (list person 'ate 'a 'good food '_ '_ '\.)
          (list person 'ate '_ 'good food '_)
          'what-answer)
    (list (list 'where 'did person 'eat food '_ '_ '?)
          (list person 'ate '_ '_ food 'where)
          'where-question)
    (list (list person 'ate food 'at restaurant '_ '_ '\.)
          (list person 'ate '_ '_ food restaurant)

```

```

'where-answer)
(list (list 'how 'did person 'like food 'at restaurant '?
      (list person 'thought '_ 'what food restaurant)
      'how-question)
(list (list person 'thought food 'was 'good 'at restaurant '\.)
      (list person 'thought '_ 'good food restaurant)
      'how-answer)
))

```

```

(defun clothing-list (person item store)
  (list
    (list (list '$shopping '$clothing person item store '_ '_))
    (list (list person 'went 'to store '_ '_ '\.)
          (list person 'went '_ '_ store)
          'entering)
    (list (list person 'looked 'for 'good item '_ '_ '\.)
          (list person 'looked '_ 'good item '_)
          'looking)
    (list (list person 'tried 'on 'several item '_ '_ '\.)
          (list person 'tried person 'several item '_)
          'deciding)
    (list (list person 'took 'the 'best item '_ '_ '\.)
          (list person 'took '_ 'best item '_)
          'taking)
    (list (list person 'paid 'the 'cashier '_ '_ '\.)
          (list person 'paid 'cashier '_ '_)
          'paying)
    (list (list person 'left store '_ '_ '\.)
          (list person 'left '_ '_ store)
          'leaving)
    ;; the clothing stories only have six sentences
    (list (list '_ '_ '_ '_ '_ '_)
          (list '_ '_ '_ '_ '_)
          'filler)
    (list (list 'who 'bought item 'at store '_ '_ '?
              (list 'who 'bought '_ '_ item store)
              'who-question)
          (list (list person 'did '_ '_ '_ '_ '\.)
                (list person 'did '_ '_ '_)
                'who-answer)
          (list (list 'what 'did person 'buy 'at store '_ '?
                    (list person 'bought '_ '_ 'what store)
                    'what-question)
          (list (list person 'bought item '_ '_ '_ '\.)
                (list person 'bought '_ '_ item '_)
                'what-answer)
          (list (list 'where 'did person 'buy item '_ '_ '?
                    (list person 'bought '_ '_ item 'where)
                    'where-question)
          (list (list person 'bought item 'at store '_ '_ '\.)
                (list person 'bought '_ '_ item store)
                'where-answer)
    ;; the clothing stories only have three questions
    (list (list '_ '_ '_ '_ '_ '_)
          (list '_ '_ '_ '_ '_)
          'filler)
    (list (list '_ '_ '_ '_ '_ '_)
          (list '_ '_ '_ '_ '_)
          'filler)
  ))

```

```

(defun plane-list (person origin dest)
  (list
    (list (list '$travel '$plane person origin dest 'big '_))
    (list (list person 'went 'to origin '_ '_ '\.)

```

```

        (list person 'went '_ '_ '_ origin)
        'origin)
    (list (list person 'checked-in 'for 'a 'flight 'to dest '\.)
          (list person 'checked-in '_ dest 'flight '_)
          'checking-in)
    (list (list person 'waited 'at 'the 'gate 'for 'boarding '\.)
          (list person 'waited '_ '_ 'boarding 'gate)
          'waiting)
    (list (list person 'got-on 'the 'plane '_ '_ '_ '\.)
          (list person 'got-on '_ '_ '_ 'plane)
          'getting-on)
    (list (list 'the 'plane 'took-off 'from origin '_ '_ '\.)
          (list 'plane 'took-off '_ '_ '_ origin)
          'taking-off)
    (list (list 'the 'plane 'arrived 'at dest '_ '_ '\.)
          (list 'plane 'arrived '_ '_ '_ dest)
          'arriving)
    (list (list person 'got-off 'the 'plane '_ '_ '_ '\.)
          (list person 'got-off '_ '_ '_ 'plane)
          'getting-off)
    (list (list 'who 'traveled 'to dest '_ '_ '_ '?')
          (list 'who 'traveled '_ '_ '_ dest)
          'who-question)
    (list (list person 'did '_ '_ '_ '_ '\.)
          (list person 'did '_ '_ '_ '_)
          'who-answer)
    (list (list 'what 'did person 'take 'to dest '_ '?')
          (list person 'took '_ '_ 'what dest)
          'what-question)
    (list (list person 'took 'a 'plane '_ '_ '_ '\.)
          (list person 'took '_ '_ 'plane '_)
          'what-answer)
    (list (list 'where 'did person 'take 'a 'plane 'to '?')
          (list person 'took '_ '_ 'plane 'where)
          'where-question)
    (list (list person 'took 'a 'plane 'to dest '_ '\.)
          (list person 'took '_ '_ 'plane dest)
          'where-answer)
    (list (list 'did person 'travel 'big 'distance 'to dest '?')
          (list person 'traveled '_ 'big 'distance dest)
          'did-question)
    (list (list person 'traveled 'a 'big 'distance '_ '_ '\.)
          (list person 'traveled '_ 'big 'distance '_)
          'did-answer)
))

```

C.1.3 Training data

The simulation file initially contains only 9 lines specifying the data files, simulation parameters and snapshot iterations to be saved. Each snapshot is saved at the end of the simulation file. A snapshot contains the current word representations and the current weights for the FGREP modules. Shown below is the simufile with only the final configuration stored, showing the first two word representations (for \$restaurant and \$fancy), and the beginning of the weights for the sentence parser module.

```

voc                ;;; wordfile
inp                ;;; inputfile
repfile           ;;; reps, if developed on separate machines
6 12 100 75 75 100 50 50    ;;; nnets, nwordrep, hidrep[neti] (DIVISIBLE)
1 1 1 1 1 1           ;;; running[neti]

```

```

0 0 1 5          ;; displaying, testing, seed, nphase
5000 10000 15000 20000 25000  ;; (+) lasttimes(epochs) for each phase
0.1 0.05 0.025 0.01 0.005    ;; eta for each phase
25000 9999999999          ;; snapshots, (epoch):-1=initial, 9x9=last
25000
0.065524 0.533577 0.006305 0.024564 0.038476 0.513618 0.000749 0.059773
0.612141 0.998208 0.170227 0.000885
...
-0.109982 0.029074 0.512288 -0.310273 0.574082 -3.896462 -1.033375 0.023748
0.514780 0.205543 -0.114771 -2.638781 ...

```

The training set (of story data) is read from the file `inp` (generated by the `gener` program (section C.1.2)). Below are a few sample stories from this file. The first line in each story contains the words which make up slot-filler representation for the story, i.e. the `nslot` words whose distributed representations are concatenated to form the representation vector.

Subsequently, `nsent` sentences follow. The number 1 at the beginning of each sentence (the `included-flag`) indicates whether this sentence is included in the input story. All sentences are always included during training. However, the same format is used for the story data during performance phase, and incomplete stories are used to test the system. The first line of the sentence specification contains the sequence of words (max `nword`), the second line specifies the case-role representation for the sentence (width: `ncase` word representations). The last word (in caps) gives a name for each event (this is just for readability and ignored by the program).

After the sentences there are the questions and answers used in the context of this story. They are given in the same format as the sentences, and they refer to the same story representation in the beginning of the story data.

```

7 8 6 7 4          ;; nslot, nword, ncase, nsent, nquest
$restaurant $fancy person fancy-food fancy-rest good big
1 person went to fancy-rest _ _ _ _ _
  person went _ _ _ fancy-rest ENTERING
1 the waiter seated person _ _ _ _ _
  waiter seated _ _ person _ SEATING
1 person asked the waiter for fancy-food _ _ _
  person asked waiter _ fancy-food _ ORDERING
1 person ate a good fancy-food _ _ _ _ _
  person ate _ good fancy-food _ EATING
1 person paid the waiter _ _ _ _ _
  person paid waiter _ _ _ PAYING
1 person left a big tip _ _ _ _ _
  person left waiter big tip _ TIPPING
1 person left fancy-rest _ _ _ _ _
  person left _ _ _ fancy-rest LEAVING
1 who ate fancy-food at fancy-rest _ _ _ _ ?
  who ate _ _ fancy-food fancy-rest WHO-QUESTION
1 person did _ _ _ _ _
  person did _ _ _ _ WHO-ANSWER
1 what did person eat at fancy-rest _ _ _ ?
  person ate _ _ what fancy-rest WHAT-QUESTION
1 person ate a good fancy-food _ _ _ _ _
  person ate _ good fancy-food _ WHAT-ANSWER
1 where did person eat fancy-food _ _ _ _ ?
  person ate _ _ fancy-food where WHERE-QUESTION
1 person ate fancy-food at fancy-rest _ _ _ _
  person ate _ _ fancy-food fancy-rest WHERE-ANSWER
1 how did person like fancy-food at fancy-rest ?
  person thought _ what fancy-food fancy-rest HOW-QUESTION
1 person thought fancy-food was good at fancy-rest _
  person thought _ good fancy-food fancy-rest HOW-ANSWER

```

```

$shopping $electronics person electr-item electr-store _ _
1 person went to electr-store _ _ _
  person went _ _ _ electr-store ENTERING
1 person looked for good electr-item _ _ _
  person looked _ good electr-item _ LOOKING
1 person asked the salesperson questions about electr-item _
  person asked salesperson electr-item questions _ DECIDING
1 person took the best electr-item _ _ _
  person took _ best electr-item _ TAKING
1 person paid the cashier _ _ _
  person paid cashier _ _ _ PAYING
1 person left electr-store _ _ _
  person left _ _ _ electr-store LEAVING
-1 _ _ _ _ _ FILLER
1 who bought electr-item at electr-store _ _ ?
  who bought _ _ electr-item electr-store WHO-QUESTION
1 person did _ _ _ _ _
  person did _ _ _ _ _ WHO-ANSWER
1 what did person buy at electr-store _ ?
  person bought _ _ what electr-store WHAT-QUESTION
1 person bought electr-item _ _ _
  person bought _ _ electr-item _ WHAT-ANSWER
1 where did person buy electr-item _ _ ?
  person bought _ _ electr-item where WHERE-QUESTION
1 person bought electr-item at electr-store _ _
  person bought _ _ electr-item electr-store WHERE-ANSWER
-1 _ _ _ _ _ FILLER
-1 _ _ _ _ _ FILLER
-1 _ _ _ _ _ FILLER

$travel $plane person plane-origin plane-dest big _
1 person went to plane-origin _ _ _
  person went _ _ _ plane-origin ORIGIN
1 person checked-in for a flight to plane-dest _
  person checked-in _ plane-dest flight _ CHECKING-IN
1 person waited at the gate for boarding _
  person waited _ _ boarding gate WAITING
1 person got-on the plane _ _ _
  person got-on _ _ _ plane GETTING-ON
1 the plane took-off from plane-origin _ _ _
  plane took-off _ _ _ plane-origin TAKING-OFF
1 the plane arrived at plane-dest _ _ _
  plane arrived _ _ _ plane-dest ARRIVING
1 person got-off the plane _ _ _
  person got-off _ _ _ plane GETTING-OFF
1 who traveled to plane-dest _ _ _ ?
  who traveled _ _ _ plane-dest WHO-QUESTION
1 person did _ _ _ _ _
  person did _ _ _ _ _ WHO-ANSWER
1 what did person take to plane-dest _ ?
  person took _ _ what plane-dest WHAT-QUESTION
1 person took a plane _ _ _
  person took _ _ plane _ WHAT-ANSWER
1 where did person take a plane to ?
  person took _ _ plane where WHERE-QUESTION
1 person took a plane to plane-dest _ _
  person took _ _ plane plane-dest WHERE-ANSWER
1 did person travel big distance to plane-dest ?
  person traveled _ big distance plane-dest DID-QUESTION
1 person traveled a big distance _ _ _
  person traveled _ big distance _ DID-ANSWER

```

Above, we have shown what the contents of the training set are. The actual input files that DISCERN reads contain word indices instead of actual words:

```

7 8 6 7 4
0 1 12 14 13 61 63      ;;: nslot, nword, ncase, nsent, nquest
1 12 36 82 13 -1 -1 -1 76 12 36 -1 -1 -1 13 ENTERING
1 78 31 37 12 -1 -1 -1 76 31 37 -1 -1 12 -1 SEATING
1 12 38 78 31 80 14 -1 76 12 38 31 -1 14 -1 ORDERING
1 12 40 77 61 14 -1 -1 76 12 40 -1 61 14 -1 EATING
1 12 42 78 31 -1 -1 -1 76 12 42 31 -1 -1 -1 PAYING
1 12 43 77 63 56 -1 -1 76 12 43 31 63 56 -1 TIPPING
1 12 43 13 -1 -1 -1 -1 76 12 43 -1 -1 -1 13 LEAVING
1 87 40 14 83 13 -1 -1 86 87 40 -1 -1 14 13 WHO-QUESTION
1 12 91 -1 -1 -1 -1 -1 76 12 91 -1 -1 -1 -1 WHO-ANSWER
1 88 91 12 92 83 13 -1 86 12 40 -1 -1 88 13 WHAT-QUESTION
1 12 40 77 61 14 -1 -1 76 12 40 -1 61 14 -1 WHAT-ANSWER
1 89 91 12 92 14 -1 -1 86 12 40 -1 -1 14 89 WHERE-QUESTION
1 12 40 14 83 13 -1 -1 76 12 40 -1 -1 14 13 WHERE-ANSWER
1 90 91 12 93 14 83 13 86 12 94 -1 88 14 13 HOW-QUESTION
1 12 94 14 95 61 83 13 76 12 94 -1 61 14 13 HOW-ANSWER

```

The numbers are indices to the word representations in the simulation file above, and they also serve as indices to the word file which contains the actual words. There are 99 words in the training vocabulary:

\$restaurant	\$fancy	\$coffee	\$fast	\$shopping
\$clothing	\$electronics	\$grocery	\$travel	\$plane
\$train	\$bus	PERSON	FANCY-REST	FANCY-FOOD
COFFEE-REST	COFFEE-FOOD	FAST-REST	FAST-FOOD	CLOTH-STORE
CLOTH-ITEM	ELECTR-STORE	ELECTR-ITEM	GROCERY-STORE	GROCERY-ITEM
PLANE-ORIGIN	PLANE-DEST	TRAIN-ORIGIN	TRAIN-DEST	BUS-ORIGIN
BUS-DEST	waiter	cashier	conductor	salesperson
driver	went	seated	asked	waited
ate	tasted	paid	left	punched
traveled	looked	tried	compared	took
checked-in	took-off	got-on	arrived	got-off
bought	tip	none	line	several
prices	good	bad	big	small
best	questions	shopping-cart	gate	flight
boarding	ticket	plane	train	bus
distance	.	a	the	about
for	from	to	at	on
in	?	who	what	where
how	did	eat	like	thought
was	buy	take	travel	

The performance of FGREP modules is tested with the complete DISCERN program, presented in appendix D. The lexicon and the hierarchical feature maps are simply turned off. This is how the performance tables in sections 5.3.4 and 5.3.5 were obtained. Samples of the test data are given in section D.3, and code for generating it is listed in section D.2.

C.2 Hierarchical feature maps

C.2.1 Training code

The hierarchical feature maps of the episodic memory (chapter 7) were organized and tested with the below simulation program. The single-level feature mapping of script-based stories (sections 6.1 and 7.3.2) and the mapping of FGREP representations of section 4.4.2 were also created using the hierarchical feature map code, only the number of levels was limited to one.

The code contains the testing routines for testing the map with noise. The graphics routines and non-essential initializations, file-I/O etc. have been omitted.

```

#include <stdio.h>
#include <math.h>

/* max table dimensions */
#define maxnets 8          /* maximum individual map size is 8*8 units */
#define maxss 10           /* maximum number of snapshots */
#define maxvectors 200     /* maximum number of input vectors */
#define maxlabell 30       /* maximum length of labels */
#define maxrep 12          /* maximum length of word representations */
#define maxslot 7          /* maximum number of assemblies in storyrep */
#define maxwords 300       /* maximum number of words */
#define maxdim (maxrep*maxslot) /*maximum dimension of input vectors */

/* data constants etc. */
#define ntoks 2             /* number of id units */
#define firsttypeword 12    /* index of first type word */
#define ntypewords 19       /* number of type words */
#define firsttokword 108    /* index of first word instance */
#define lownoise 0.0        /* lowest noise level tested */
#define highnoise 1.01      /* upper limit for the noise level */
#define noisestep 0.05      /* noise increment */
#define err 0.15            /* statistics collected within this error */
#define snapshotend 99999999 /* token that indicates last snapshot in file*/

int nets, t1, t2, nct0, nct1, /* top level: netsize,phaseends,neighborhoods*/
    mnets, mt0, mt1, mt2, mnct0, mnct1, /* middle level */
    bnets, bt0, bt1, bt2, bnct0, bnct1, /* bottom level */
    besti, bestj, tbesti, tbestj, mbesti,mbestj, bbesti, bbestj, /*image indices*/
    nvector, nslot, nwords, nrep, dim /* dimensions */
    seed, startepoch, lastepoch, nextsnapshot, snapshots[maxss], /* simulation */
    fildes, cmapsize, displaying, continuing, testing, /* parameters */

/* computing variance on the input lines */
float sums[maxnets][maxnets][maxdim],
    square_sums[maxnets][maxnets][maxdim],
    msums[maxnets][maxnets][maxnets][maxnets][maxdim],
    msquare_sums[maxnets][maxnets][maxnets][maxnets][maxdim],
    minvar0, minvar1;
int ns[maxnets][maxnets], mns[maxnets][maxnets][maxnets][maxnets];

/* compression */
int headindices[maxdim], indices[maxnets][maxnets][maxdim],
    mindices[maxnets][maxnets][maxnets][maxnets][maxdim],
    nmlines[maxnets][maxnets],
    nblines[maxnets][maxnets][maxnets][maxnets];

/* statistics variables */
int ncorr,ntokcorr,ntokall,nerr,
    nmapcorr[maxnets][maxnets][maxnets][maxnets],
    nmapvectors[maxnets][maxnets][maxnets][maxnets],
    ntokmapcorr[maxnets][maxnets][maxnets][maxnets],
    ntokmapall[maxnets][maxnets][maxnets][maxnets],
    nmaperr[maxnets][maxnets][maxnets][maxnets];
float mapdeltasum[maxnets][maxnets][maxnets][maxnets];

/* word representations */
float blankrep[maxrep]; /* blankrep=reps[-1] */
float reps[maxwords][maxrep];

float image[maxdim], /* vector retrieved from the map */
    deltasum, inpvector[maxdim];
float noise; /* noise level */
int shuffletable[maxinps]; /* table of indices in this epoch */

```



```

/* learning rates */
float alphas0,alphas1, malphas0, malphas1, balphas0, balphas1;

/* file names */
char simufile[100], inputfile[100], testfile[100], repfile[100];
FILE *fp;

/* story representations and labels */
struct inputvectors {
    char label[maxlabel+1];
    int index[maxslot];
} story[maxvectors];

/* units at the three levels of maps */
struct unitdata {
    int labelcount;
    int labels[maxvectors+1];
    float bestvalue;
    float value;
    float comp[maxdim];
} units[maxnets][maxnets],
  munits[maxnets][maxnets][maxnets][maxnets],
  bunits[maxnets][maxnets][maxnets][maxnets][maxnets][maxnets];

/***** main control *****/

/* the simulation control functions main(), iterate_snapshots()
   and training() are similar to ones listed in C.1.1 and have been omitted */

iterate_inputs()
{
    register int i;
    init_units(); /* (not listed) clear the statistics */
    for(i=0; i<nvectors; i++)
        story(shuffletable[i]);
    if (t==mt0-1)
        choose_indices(); /* compress middle level input lines */
    if (t==bt0-1)
        choose_mindices(); /* compress bottom level input lines */
    print_stats();
}

story(storyi)
/* present one input story */
int storyi;
{
    if (!testing) /* set up IDs randomly for each story */
        randomize_tokens();
    form_inpvector(storyi);
    /* present the input to the top level */
    present_input(units,nets,storyi,
        t,t1,t2,nct0,nct1,alphas0,alphas1,dim,headindices);
    tbesti=besti; /* top level image unit indices */
    tbestj=bestj;
    if (t==mt0-1) /* cumulate variance */
        cumulate_sums();
    if ( t>=mt0 )
    {
        /* present the input to the middle level */
        present_input(munits[tbesti][tbestj], mnets, storyi,
            t-mt0, mt1-mt0, mt2-mt0, mnct0, mnct1,
            malphas0, malphas1,nmlines[tbesti][tbestj],
            indices[tbesti][tbestj]);
        mbesti=besti; /* middle level image unit indices */
        mbestj=bestj;
        if (t==bt0-1)
            cumulate_msums(); /* cumulate variance */
        if ( t>=bt0 )

```

```

    {
        /* present the input to the bottom level */
        present_input(bunits[tbesti][tbestj][mbesti][mbestj],
                    bnets, storyi,
                    t-bt0, bt1-bt0, bt2-bt0, bnct0, bnct1,
                    balphat0, balphat1,
                    nblines[tbesti][tbestj][mbesti][mbestj],
                    mindices[tbesti][tbestj][mbesti][mbestj]);
        bbesti=besti; /* bottom level image unit indices */
        bbestj=bestj;
    }
}
get_image(storyi);
collect_stats(storyi);
}

present_input(table,nets,input,t,t1,t2,nct0,nct1,alphat0,alphat1,dim,indices)
/* present input to the map and adapt it */
struct unitdata table[maxnets][maxnets];
int nets,input,t,t1,t2,nct0,nct1,dim,indices[];
float alphat0,alphat1;
{
    float alpha();
    compute_responses(table,nets,input,dim,indices);
    if (!testing)
        modify_weights(table,nets,
                    nc(t, t1, t2, nct0, nct1),
                    alpha(t, t1, t2, alphat0, alphat1),
                    dim,indices);
}

/***** feature map response and adaptation *****/

compute_responses(table, nets, inpv,dim,indices)
/* compute responses of all units and find the maximally responding one */
struct unitdata table[maxnets][maxnets];
int inpv, nets,dim,indices[];
{
    register int i,j;
    float seldistance(); /* see math routines in section D.1.2 */
    float best=999999999.9;
    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
            {
                table[i][j].value =
                    seldistance(indices,inpvector, table[i][j].comp, dim);
                /* check if this unit's response is best so far encountered */
                if (table[i][j].value < best)
                    {
                        besti=i; bestj=j; best=table[i][j].value;
                    }
            }
}

modify_weights(table, nets, nc, alpha, dim,indices)
/* modify weighs toward the input in a neighborhood of the maximally
   responding unit */
struct unitdata table[maxnets][maxnets];
int nets, nc, dim,indices[];
float alpha;
{
    register int i,j,k;
    for(i=besti-nc; i<=besti+nc; i++)
        for(j=bestj-nc; j<=bestj+nc; j++)
            if (i>=0 && i<nets && j>=0 && j<nets)
                /* modify weighs toward the input; equation 6.6 */
}

```

```

        for(k=0; k<dim; k++)
            table[i][j].comp[k] +=
                alpha*(inpvector[indices[k]]-table[i][j].comp[k]);
    }

    /***** compression *****/

    cumulate_sums()
    /* cumulate info for variance calculation at the top level lines */
    {
        register int i,j;
        ns[tbesti][tbestj]++;
        for (i=0; i<dim; i++)
            {
                sums[tbesti][tbestj][i] += inpvector[i];
                square_sums[tbesti][tbestj][i] += inpvector[i]*inpvector[i];
            }
    }

    cumulate_msums()
    /* cumulate info for variance calculation at the middle level lines */
    {
        register int i,j;
        mns[tbesti][tbestj][mbesti][mbestj]++;
        for (i=0; i<nmlines[tbesti][tbestj]; i++)
            {
                msums[tbesti][tbestj][mbesti][mbestj][i] +=
                    inpvector[indices[tbesti][tbestj][i]];
                msquare_sums[tbesti][tbestj][mbesti][mbestj][i] +=
                    inpvector[indices[tbesti][tbestj][i]]*
                    inpvector[indices[tbesti][tbestj][i]];
            }
    }

    choose_indices()
    /* determine the input lines to be passed to the middle level */
    {
        register int i,j,k,kk;
        int bestkk;
        float best;
        /* compute variance for each line */
        for (i=0; i<nets; i++)
            for (j=0; j<nets; j++)
                for (k=0; k<dim; k++)
                    if (ns[i][j]>0)
                        sums[i][j][k] = square_sums[i][j][k]-
                            sums[i][j][k]*sums[i][j][k]/ns[i][j];
                    else
                        sums[i][j][k] = 0.0;
        for (i=0; i<nets; i++)
            for (j=0; j<nets; j++)
                {
                    /* find all lines with variance > minvar0 */
                    best=99999999.9;
                    for (k=0; k<dim && best>minvar0; k++)
                        {
                            bestkk=0;
                            best=(-1.0);
                            for (kk=0; kk<dim; kk++)
                                if (sums[i][j][kk]>best)
                                    {
                                        best=sums[i][j][kk];
                                        bestkk=kk;
                                    }
                            /* collect their indices */
                            indices[i][j][k]=bestkk;
                            sums[i][j][bestkk]=(-1.0);
                        }
                }
    }

```

```

    }
    /* number of input lines to middle level */
    if(best<=minvar0)
        nmlines[i][j]=k-1;
    else
        nmlines[i][j]=k;
    printf("Unit %d, %d passes %d lines\n",i,j,nmlines[i][j]);
}
}

```

```

choose_mindices()
/* determine the input lines to be passed to the bottom level */
{
    register int i,j,ii,jj,k,kk;
    int bestkk;
    float best;
    /* compute variance for each line */
    for (i=0; i<nets; i++)
        for (j=0; j<nets; j++)
            for (ii=0; ii<mnets; ii++)
                for (jj=0; jj<mnets; jj++)
                    for (k=0; k<nmlines[i][j]; k++)
                        if (mns[i][j][ii][jj]>0)
                            msums[i][j][ii][jj][k] = msquare_sums[i][j][ii][jj][k]
                                -msums[i][j][ii][jj][k]*msums[i][j][ii][jj][k]/
                                    mns[i][j][ii][jj];
                        else
                            msums[i][j][ii][jj][k] = 0.0;
    for (i=0; i<nets; i++)
        for (j=0; j<nets; j++)
            for (ii=0; ii<mnets; ii++)
                for (jj=0; jj<mnets; jj++)
                    {
                        /* find all lines with variance > minvar0 */
                        best=999999999.9;
                        for (k=0; k<nmlines[i][j] && best>minvar1; k++)
                            {
                                bestkk=0;
                                best=(-1.0);
                                for (kk=0; kk<nmlines[i][j]; kk++)
                                    if (msums[i][j][ii][jj][kk]>best)
                                        {
                                            best=msums[i][j][ii][jj][kk];
                                            bestkk=kk;
                                        }
                                /* collect their indices */
                                mindices[i][j][ii][jj][k]=indices[i][j][bestkk];
                                asums[i][j][ii][jj][bestkk]=(-1.0);
                            }
                        /* number of input lines to bottom level */
                        if(best<=minvar1)
                            nblines[i][j][ii][jj]=k-1;
                        else
                            nblines[i][j][ii][jj]=k;
                        printf("Unit %d, %d, %d, %d passes %d lines\n",
                            i,j,ii,jj,nblines[i][j][ii][jj]);
                    }
}
}

```

```

/***** statistics *****/

```

```

collect_stats(input)
int input;
{
    register int i;
    float uniterr;

```

```

nmapvectors[tbesti][tbestj][mbesti][mbestj]++;
for(i=0; i<dim; i++)
{
    uniterr = fabs(image[i] - reps[ story[input].index[i/nrep] ][i%nrep]);
    /* cumulate data for counting units within error range */
    if(uniterr<err)
    {
        nerr++;
        nmaperr[tbesti][tbestj][mbesti][mbestj]++;
    }
    /* cumulate data for average error per unit */
    deltasum += uniterr;
    mapdeltasum[tbesti][tbestj][mbesti][mbestj] += uniterr;
}
/* cumulate data for counting the number of words nearest to the correct
lexicon entry */
for(i=0; i<nslot; i++)
{
    /* cumulate the total word instance counters */
    if(story[input].index[i]>=firsttokword)
    {
        ntokmapall[tbesti][tbestj][mbesti][mbestj]++;
        ntokall++;
    }
    /* determine_nearest is similar to find_nearest, see D.1.2 */
    if(determine_nearest(&image[i*nrep]) == story[input].index[i])
    {
        ncorr++;
        nmapcorr[tbesti][tbestj][mbesti][mbestj]++;
        /* cumulate the word instance counters */
        if(story[input].index[i]>=firsttokword)
        {
            ntokcorr++;
            ntokmapcorr[tbesti][tbestj][mbesti][mbestj]++;
        }
    }
}
}
}

print_stats()
{
    int i,j,ii,jj;
    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
            for(ii=0; ii<nets; ii++)
                for(jj=0; jj<nets; jj++)
                    if(nblines[i][j][ii][jj]>0)
                    {
                        if(nmapvectors[i][j][ii][jj]==0) nmapvectors[i][j][ii][jj]=1;
                        if(ntokmapall[i][j][ii][jj]==0) ntokmapall[i][j][ii][jj]=1;
                        printf("Net %d,%d,%d,%d: %6.1f %6.1f %9.4f", i,j,ii,jj,
                            100.0*nmapcorr[i][j][ii][jj]/
                                (nslot*nmapvectors[i][j][ii][jj]),
                            100.0*ntokmapcorr[i][j][ii][jj]/
                                ntokmapall[i][j][ii][jj],
                            100.0*nmaperr[i][j][ii][jj]/
                                (dim*nmapvectors[i][j][ii][jj]),
                            mapdeltasum[i][j][ii][jj]/
                                (dim*nmapvectors[i][j][ii][jj]));
                        printf(" %d/%d %d/%d %d %f\n",
                            nmapcorr[i][j][ii][jj], nmapvectors[i][j][ii][jj],
                            ntokmapcorr[i][j][ii][jj], ntokmapall[i][j][ii][jj],
                            nmaperr[i][j][ii][jj], mapdeltasum[i][j][ii][jj]);
                    }
}

if(ntokall==0) ntokall=1;
printf("noise %.2f: %6.1f %6.1f %9.4f", noise,

```

```

        100.0*ncorr/(nslot*nvectors),
        100.0*ntokcorr/ntokall,
        100.0*nerr/(dim*nvectors),
        deltasum/(dim*nvectors));
printf("   %d %d %d %d %d %f\n", ncorr, nvectors, tokcrr, ntokall,
       nerr, deltasum);
}

/***** simulation setup *****/

randomize_tokens()
{
    register int i,j;

    for(j=0; j<ntoks; j++)
        /* the type words form a continuous block in the lexicon */
        for(i=firsttypeword; i<firsttypeword+ntypewords; i++)
            randfun(&reps[i][j], 0.0, 1.0);
}

form_inpvector(inp)
/* corrupt the input vector with noise */
int inp;
{
    register int i,j;
    float f01rnd();

    for(i=0; i<nslot; i++)
        for(j=0; j<nrep; j++)
            inpvector[i*nrep+j] = (1.0-noise)*reps[ story[inp].index[i] ][j]
            +noise*f01rnd();
}

get_image(input)
/* collect the accurate (compressed) input lines from all levels */
int input;
{
    int i;

    for(i=0; i<dim; i++)
        image[i] = units[tbesti][tbestj].comp[i];
    /* replace top-level weights with more accurate weights at middle level */
    if(t>=mt0 && mnets>0)
        for (i=0; i<nmlines[tbesti][tbestj]; i++)
            image[ indices[tbesti][tbestj][i] ] =
                munits[tbesti][tbestj][mbesti][mbestj].comp[i];
    /* replace these weights with the accurate weights at the bottom level */
    if(t>=bt0 && bnets>0)
        for (i=0; i<nblines[tbesti][tbestj][mbesti][mbestj]; i++)
            image[ mindices[tbesti][tbestj][mbesti][mbestj][i] ] =
                bunits[tbesti][tbestj][mbesti][mbestj][bbesti][bbestj].comp[i];
}

float alpha(t,t1,t2,alpat0,alpat1)
int t,t1,t2;
float alpat0,alpat1;
{
    /* decrease gain linearly from alpat0 to alpat1 during the period 0-t1
       and linearly from alpat1 to 0 during the period t1-t2 */
    if (t<t1) return(alpat0-t*(alpat0-alpat1)/t1);
    else return(alpat1-(t-t1)*alpat1/(t2-t1));
}

nc(t,t1,t2,nct0,nct1)
int t,t1,t2,nct0,nct1;
{
    /* decrease neighborhood size linearly from nct0 to nct1+1 during

```

```

    the period 0-t1 and from nct1 to 0 during the period t1-t2 */
    if (t<t1) return(floor(nct0+0.999999-t*(0.0+nct0-nct1)/t1));
    else return(floor(nct1+0.999999-(t-t1)*(nct1+0.999999)/(t2-t1)));
}

float f01rnd()
{
    /* random float between [0,1] */
    return (rand()/32767.0);
}

```

C.2.2 Training data

The hierarchical feature maps of the episodic memory were organized using the final semantic word representations, developed by the FGREP modules (see previous section).

The simulation is specified and saved in a file similar to the one used for developing FGREP modules. Initially the file specifies input data files and simulation parameters. Simulation snapshots are appended to this file.

Below is the simulation file for the hierarchical feature maps in DISCERN. The snapshot consists of the unit weights and the indices of the input lines that the unit passes down to its submap. The first twelve weights and the first ten lines of unit (0,0) are shown below. Unit (0,0) stands for the restaurant script, and \$restaurant is accurately represented in the first twelve weights of this unit (compare to the top of the FGREP simulation file, section C.1.3). The input lines with the highest variance belong to R/taste (62,63), track (15,22), R/tip (76,82,79,74), R/food (38) and R/restaurant (53).

```

reps          ;;: representations
hfminp        ;;: inputfile
testing       ;;: testfile (not used)
0 0 12345 1000 0.001 0.0005 ;;: displaying,testing,seed,minvar0,minvar1
2           4 1000 1 0 0.02 0.01 ;;: t: nets, t1,t2,nct0,nct1,alphan0,alphan1
2 6         10 1000 1 0 0.02 0.01 ;;: m: nets,t0,t1,t2,nct0,nct1,alphan0,alphan1
8 12 100 1000 6 1 0.2 0.1 ;;: b: nets,t0,t1,t2,nct0,nct1,alphan0,alphan1
1000 9999999999 ;;: snapshots (epoch):-1=initial, 9x9=last
1000
0.065524 0.533571 0.006305 0.024564 0.038476 0.513612 0.000749 0.059773
0.612135 0.998200 0.170227 0.000885
...
62 63 15 76 82 79 74 38 22 53 ...
...

```

The representation file simply contains the representation vectors, taken from the FGREP simulation file (section C.1.3). There are 99 representations 12 components each. The first vector below stands for \$restaurant, the second one for PERSON. The ID part (first two units) of PERSON is randomly set for each input (with function `randomize_tokens`); the file shows the last pattern during the FGREP training.

```

99 12
0.065524 0.533577 0.006305 0.024564 0.038476 0.513618 0.000749 0.059773
0.612141 0.998206 0.170227 0.000885
...
0.986663 0.017304 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
...

```

The input file consists of the slot filler representations for each story, samples of which are given below:

```

7           ;;; nslot
RFB
$restaurant $fancy person fancy-food fancy-rest good big
SE
$shopping $electronics person electr-item electr-store _ _
TPB
$travel $plane person plane-origin plane-dest big _
...

```

The labels in caps identify the story representation, for example RFB = fancy-restaurant, big tip. Again, in reality this file contains word indices instead of words:

```

7           ;;; nslot
RFB
0 1 12 14 13 61 63
SE
4 6 12 22 21 -1 -1
TPB
8 9 12 25 26 63 -1
...

```

The hierarchical feature maps are tested with the same program that develops them, but different input data is used. The representation file now contains representations for the word instances instead of prototype words. The 19 prototypes have been replaced with two instances each, resulting in 118 representations total. For example, PERSON has been replaced by John (with IDs 0.75, 0.25) and Mary (with 0.25, 0.75):

```

118 12
0.065524 0.533577 0.006305 0.024564 0.038476 0.513618 0.000749 0.059773
0.612141 0.998208 0.170227 0.000885
...
0.750000 0.250000 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
0.250000 0.750000 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
...

```

Also the input file specifies story representations which consist of word instances.

```

7           ;;; nslot
RFJLMB
$restaurant $fancy john lobster mamaison good big
SEMCR
$shopping $electronics mary cd-player radioshack _ _
TPJLS
$travel $plane john lax sfo big _

```

C.3 Lexicon

C.3.1 Training code

The DISCERN lexicon (chapter 9) is organized and its accuracy is tested with the program below. The lexicon for the sentence case role assignment data in section 9.3 is also developed with the same program, but with different data.

Again, graphics, initialization, and file I/O are omitted.


```

#include <stdio.h>
#include <math.h>

/* table dimensions */
#define maxnets 20          /* maximum network size is n*n units */
#define maxsnaps 50        /* maximum number of snapshots */
#define maxvectors 200     /* maximum number of input vectors */
#define maxinps 200       /* maximum number of inputs */
#define maxlabell 30      /* maximum length of labels */
#define maxdim 50         /* maximum dimension of input vectors */

/* constants and simulation parameters */
#define firsttokword 108   /* index of first word instance */
#define ntoks 19          /* number of word instances */
#define lownoise 0.0      /* lowest noise level tested */
#define highnoise 1.01    /* upper limit for the noise level */
#define noisestep 0.05    /* noise increment */
#define lerr 0.15         /* statistics collected within this error */
#define serr 0.15        /* statistics collected within this error */
#define maxthrough 4     /* propagate through 4 strongest units */
#define snapshotend 99999999

int lnets, lt0, lt1, lt2, lnct0, lnct1, /*lex netsize,phaseends,neighborhoods*/
    lbesti, lbestj, lnvectors, ldim, /* lex image indices, #ofwords, dimension */
    snets, st0, st1, st2, snct0, snct1, /*sem netsize,phaseends,neighborhoods*/
    sbesti, sbestj, snvectors, sdim, /* sem image indices, #ofwords, dimension */
    at0, at1, at2, /* assoc netsize, phaseends */
    seed, t, startepoch, lastepoch, ninppairs, /* simulation parameters */
    displaying, continuing, testing,
    nextsnapshot, snapshots[maxsnaps];
int shuffletable[maxinps]; /* table of indices in this epoch */
float noise; /* noise level */

/* learning rates */
float lalphat0, lalphat1, salphat0, salphat1, aalphat0, aalphat1;

/* statistics variables */
int lnerr, snerr, lsner, slnerr,
    lncorr, snccorr, lsncorr, slnccorr,
    lntokcorr, sntokcorr, lsntokcorr, slntokcorr,
    lninp, sninp, aninp,
    lntokinp, sntokinp, antokinp;
float ldeltasum, sdeltasum, lsdeltasum, sldeltasum;

/* file names */
char simufile[100], inpfiler[100], lvectorfiler[100], llabelfiler[100],
    svectorfiler[100], slabelfiler[100];
FILE *fp, *fp2;

/* lexical item index and its corresponding meaning index */
struct pair {
    int l,s;
}ninppairs[maxinps];

/* lexical and semantic input vectors */
struct inputvectors {
    char label[maxlabell+1];
    float comp[maxdim];
};
struct inputvectors lvector[maxvectors], svector[maxvectors];

/* units on the feature maps */
struct unitdata {
    int labelcount;
    int labels[maxvectors+1];
    float bestvalue;
    float value;
    float comp[maxdim];
};

```

```

struct unitdata lunits[maxnets][maxnets], sunits[maxnets][maxnets];
/* associative connections */
float lsassoc[maxnets][maxnets][maxnets][maxnets],
  slassoc[maxnets][maxnets][maxnets][maxnets];
/***** main control *****/
/* the simulation control functions main(), iterate_snapshots()
   and training() are similar to ones listed in C.1.1 and have been omitted */
iterate_inputs()
{
  register int i;
  float alpha();
  init_units();
  for(i=0; i<ninppairs; i++) /* (not listed) clear the statistics */
  {
    compute_responses(shuffletable[i]);
    if (!testing)
      /* see section C.2.1 for definitions of nc() and alpha() */
      modify_weights(nc(t, lt0, lt1, lt2, lnct0, lnct1),
                    alpha(t, lt0, lt1, lt2, lalphat0, lalphat1),
                    nc(t, st0, st1, st2, snct0, snct1),
                    alpha(t, st0, st1, st2, salphat0, salphat1),
                    alpha(t, at0, at1, at2, aalphat0, aalphat1),
                    shuffletable[i]);
  }
  print_stats();
}

/***** feature map responses *****/
compute_responses(inpv)
int inpv;
{
  int lsbesti, lsbestj, slbesti, slbestj;
  /* lexical map */
  if(t>=lt0 && inppairs[inpv].l > (-1))
  {
    find_max_resp(ldim, lnets, lerr, inppairs[inpv].l, lnvectors, lunits, lvectors,
                  &lsbesti, &lsbestj,
                  &ldeltasum, &lnerr, &lnccorr, &lnctokcorr);
    find_max_assoc(lnets, snets, lbesti, lbestj, lunits, lsassoc,
                  &lsbesti, &lsbestj);
  }
  /* semantic map */
  if(t>=st0 && inppairs[inpv].s > (-1))
  {
    find_max_resp(sdim, snets, serr, inppairs[inpv].s, snvectors, sunits, svectors,
                  &sbesti, &sbestj,
                  &sdeltasum, &snerr, &snccorr, &snctokcorr);
    find_max_assoc(snets, lnets, sbesti, sbestj, sunits, slassoc,
                  &sbesti, &sbestj);
  }
  /* check associative connections */
  if(t>=lt0 && inppairs[inpv].l > (-1) &&
     t>=st0 && inppairs[inpv].s > (-1) &&
     t>=at0)
  {
    collect_stats(sdim, snvectors, serr, lsbesti, lsbestj, inppairs[inpv].s,
                  inppairs[inpv].l, sunits, svectors,
                  &lnccorr, &lnerr, &ldeltasum, &lnctokcorr);
    collect_stats(ldim, lnvectors, lerr, slbesti, slbestj, inppairs[inpv].l,
                  inppairs[inpv].s, lunits, lvectors,
                  &lnccorr, &lnerr, &sdeltasum, &snctokcorr);
  }
}

```

```

}

find_max_resp(dim,nets,err,index,nvectors, units,vectors,
              besti,bestj, deltasum,nerr,ncorr,ntokcorr)
/* find the maximally responding unit */
struct unitdata units[maxnets][maxnets];
struct inputvectors vectors[maxvectors];
int dim,nets,index,nvectors,*besti,*bestj,*nerr,*ncorr,*ntokcorr;
float err,*deltasum;
{
  int i,j;
  float inpvector[maxdim], best=999999999.9;
  float distance(),f01rnd(); /* see sections D.1.2 and C.2.1 */

  /* form the out vector */
  for(i=0; i<dim; i++)
    inpvector[i]=(1.0-noise)*vectors[index].comp[i]+
      noise*f01rnd();
  /* present it and find the image unit */
  for(i=0; i<nets; i++)
    for(j=0; j<nets; j++){
      units[i][j].value = distance(inpvector,units[i][j].comp,dim);
      /* check if this unit's response is best so far encountered */
      if (units[i][j].value < best) {
        *besti=i; *bestj=j; best=units[i][j].value;
      }
    }
  collect_stats(dim,nvectors,err,*besti,*bestj,index,index,
              units,vectors,ncorr,nerr,deltasum,ntokcorr);
}

find_max_assoc(nets,anets,besti,bestj,units,assoc,abesti,abestj)
/* propagate through the associative connections and find max assoc unit */
int nets,anets,besti,bestj,*abesti,*abestj;
struct unitdata units[maxnets][maxnets];
float assoc[maxnets][maxnets][maxnets][maxnets];
{
  float best,abest,highest,second,third,fourth;
  int i,j,ii,jj,lowi, lowj, highi, highj,
  secondi,secondj,thirdi,thirdj,fourthi,fourthj,anc=1;
  struct unitdata aunits[maxnets][maxnets];

  if(t>=at0)
  {
    for(i=0; i<anets; i++)
      for(j=0; j<anets; j++)
        aunits[i][j].value=0.0;

    /* set up the neighborhood */
    if (besti-anc>=0) lowi=besti-anc; else lowi=0;
    if (besti+anc<nets) highi=besti+anc; else highi=nets-1;
    if (bestj-anc>=0) lowj=bestj-anc; else lowj=0;
    if (bestj+anc<nets) highj=bestj+anc; else highj=nets-1;

    /* find the four most active units in the neighborhood */
    fourth=third=second=best=999999999.9;
    highest=(-1.0);
    for(i=lowi; i<=highi; i++)
      for(j=lowj; j<=highj; j++)
        {
          if (units[i][j].value < best)
          {
            fourthi=thirdi; fourthj=thirdj; fourth=third;
            thirdi=secondi; thirdj=secondj; third=second;
            secondi=besti; secondj=bestj; second=best;
            besti=i; bestj=j; best=units[i][j].value;
          }
        }
  }
}

```

```

    }
    else if (units[i][j].value < second)
    {
        fourthi=thirdi; fourthj=thirdj; fourth=third;
        thirdi=secondi; thirdj=secondj; third=second;
        secondi=i; secondj=j; second=units[i][j].value;
    }
    else if (units[i][j].value < third)
    {
        fourthi=thirdi; fourthj=thirdj; fourth=third;
        thirdi=i; thirdj=j; third=units[i][j].value;
    }
    else if (units[i][j].value < fourth)
    {
        fourthi=i; fourthj=j; fourth=units[i][j].value;
    }
    if(units[i][j].value>highest)
        highest=units[i][j].value;
}

/* propagate through these units to the output map;
equations 9.1 and 9.3 (the sigmoid has been dropped because
we are only interested in finding the max unit, not its actual value */
for(i=lowi; i<=highi; i++)
for(j=lowj; j<=highj; j++)
{
    units[i][j].value=(1.0-(units[i][j].value-best)/(highest-best));
    if((i==besti && j==bestj) ||
        (i==secondi && j==secondj) ||
        (i==thirdi && j==thirdj) ||
        (i==fourthi && j==fourthj))
        for(ii=0; ii<anets; ii++)
            for(jj=0; jj<anets; jj++)
                aunits[ii][jj].value +=units[i][j].value*assoc[i][j][ii][jj];
}

/* find the image unit in the output map */
abest=(-1.0);
for(ii=0; ii<anets; ii++)
for(jj=0; jj<anets; jj++)
    if(aunits[ii][jj].value>abest)
    {
        *abesti=ii; *abestj=jj; abest=aunits[ii][jj].value;
    }
}

/***** weight adaptation *****/
modify_weights(lnc, lalpha, sncl, salpha, aalpha, inpv)
/* adapt both maps and the associative connections between them */
int lnc, sncl, inpv;
float lalpha, salpha, aalpha;
{
    if(t>=lt0 && t<=lt2 && inppairs[inpv].l > (-1))
        modify_input_weights(lbesti,lbestj,lnc,lnets,ldim,
            lunits,lalpha,lvectors,inppairs[inpv].l);
    if(t>=st0 && t<=st2 && inppairs[inpv].s > (-1))
        modify_input_weights(sbesti,sbestj,sncl,snets,sdim,
            sunits,salpha,svectors,inppairs[inpv].s);
    if(t>=lt0 && t>=st0 && t>=at0 && t<=at2
        && inppairs[inpv].l > (-1) && inppairs[inpv].s > (-1))
    {
        modify_assoc_weights(lnc,sncl,lbesti,lbestj,sbesti,sbestj,
            lnets,snets,lunits,sunits,lsassoc,aalpha);
        modify_assoc_weights(snc,lnc,sbesti,sbestj,lbesti,lbestj,

```

```

        snets,lnets,sunits,lunits,slassoc,aalpha);
    }
}

modify_input_weights(bestj,nc,nets,dim,units,alpha,vectors,index)
/* adapt the input weights of units in the neighborhood of image unit */
int bestj,nc,nets,dim,index;
float alpha;
struct unitdata units[maxnets][maxnets];
struct inputvectors vectors[maxvectors];
{
    int i,j,k;
    for(i=bestj-nc; i<=bestj+nc; i++)
        for(j=bestj-nc; j<=bestj+nc; j++)
            if (i>=0 && i<nets && j>=0 && j<nets)
                for(k=0; k<dim; k++)
                    /* modify weighs toward the input; equation 6.6 */
                    units[i][j].comp[k] += alpha*
                    (vectors[index].comp[k] - units[i][j].comp[k]);
}

modify_assoc_weights(nc,anc,bestj,abestj,nets,anets,
                    units,aunits,assoc,aalpha)
/* adapt the associative connections */
int nc,anc,bestj,abestj,nets,anets;
float aalpha,assoc[maxnets][maxnets][maxnets][maxnets];
struct unitdata units[maxnets][maxnets],aunits[maxnets][maxnets];
{
    int i,j,ii,jj;
    float sum;
    for(i=bestj-nc; i<=bestj+nc; i++)
        for(j=bestj-nc; j<=bestj+nc; j++)
            if (i>=0 && i<nets && j>=0 && j<nets)
                {
                    for(ii=abestj-anc; ii<=abestj+anc; ii++)
                        for(jj=abestj-anc; jj<=abestj+anc; jj++)
                            if (ii>=0 && ii<anets && jj>=0 && jj<anets)
                                /* modify the associative connections through
                                Hebbian learning; equation 9.2 */
                                assoc[i][j][ii][jj] += aalpha*
                                units[i][j].value * aunits[ii][jj].value;
                    /* normalize the associative output connections of a unit */
                    sum = 0.0;
                    for(ii=0; ii<anets; ii++)
                        for(jj=0; jj<anets; jj++)
                            sum += assoc[i][j][ii][jj]*assoc[i][j][ii][jj];
                    sum = sqrt(sum);
                    for(ii=0; ii<anets; ii++)
                        for(jj=0; jj<anets; jj++)
                            assoc[i][j][ii][jj] /= sum;
                }
}

/***** statistics *****/

collect_stats(dim,nvectors,err,bestj,index,aindex,units,vectors,
             ncorr,nerr,deltasum,ntokcorr)
int dim,nvectors,bestj,index,aindex,*ncorr,*nerr,*ntokcorr;
float err,*deltasum;
struct unitdata units[maxnets][maxnets];
struct inputvectors vectors[maxvectors];
{
    int i;
    for(i=0; i<dim; i++)
        {
            /* cumulate average error */
            *deltasum += fabs(units[bestj][bestj].comp[i] - vectors[index].comp[i]);
        }
}

```

```

/* cumulate number of units within err */
if(fabs(units[besti][bestj].comp[i]-vectors[index].comp[i])<err)
  (*nerr)++;
}
/* cumulate correct word and correct instance count */
/* determine_nearest is similar to find_nearest, see D.1.2 */
if(determine_nearest(units[besti][bestj].comp,vectors,nvectors,dim) == index)
{
  (*ncorr)++;
  if(aindex>=firsttokword && aindex<firsttokword+ntoks) (*ntokcorr)++;
}
}

print_stats()
{
  if(lntokinp==0) lntokinp=1;
  if(sntokinp==0) sntokinp=1;
  if(antokinp==0) antokinp=1;
  printf("Epoch %d, noise %.2f: l: %5.1f %5.1f %5.1f %6.4f
         s: %5.1f %5.1f %5.1f %6.4f\n",
         t, noise,
         100.0*lnccorr/lninp, 100.0*lntokcorr/lntokinp,
         100.0*lnerr/(ldim*lninp), ldeltasum/(ldim*lninp),
         100.0*sncorr/sninp, 100.0*sntokcorr/sntokinp,
         100.0*sncorr/(sdim*sninp), sdeltasum/(sdim*sninp));
  printf("         sl: %5.1f %5.1f %5.1f %6.4f
         ls: %5.1f %5.1f %5.1f %6.4f\n",
         100.0*slncorr/aninp, 100.0*slntokcorr/antokinp,
         100.0*slnerr/(ldim*aninp), sldeltasum/(ldim*aninp),
         100.0*lsncorr/aninp, 100.0*lsntokcorr/antokinp,
         100.0*lsnerr/(sdim*aninp), lsdeltasum/(sdim*aninp));
}

```

C.3.2 Generating training data

The lexical representations are generated according to the darkness values of the letters in the Macintosh Geneva font, using the LISP program below.

```

(setq voc '(
JOHN          MARY          MAMAISON          LEONE\`S          LOBSTER
STEAK         DENNY\`S          NORMS             SPAGHETTI        FISH
MCDONALD\`S   BURGERKING        HAMBURGER        FRIES            BROADWAY
BULLOCK\`S    SHOES             JEANS            RADIOSHACK       CIRCUITCTY
CD-PLAYER     TV                RALPH\`S         SAFEWAY          SOFTDRINKS
VEGETABLES    LAX              DFW              JFK              SFO
CENTRALSTA    DOWNTOWNST       NEWYORK          BOSTON          BUS-TERMIN
BUS-STOP      DOWNTOWN         BEACH            WAITER           CASHIER
CONDUCTOR     STAFF            DRIVER           WENT             SEATED
ASKED         WAITED           ATE              TASTED           PAID
LEFT          PUNCHED          TRAVELED         LOOKED           TRIED
COMPARED      TOOK             CHECKED-IN       TOOK-OFF        GOT-ON
ARRIVED       GOT-OFF          BOUGHT           TIP              NONE
LINE          SEVERAL          PRICES           GOOD             BAD
BIG           SMALL            BEST             QUESTIONS        CART
GATE          FLIGHT           BOARDING         TICKET           PLANE
TRAIN         BUS              DISTANCE         \.              A
THE           ABOUT            FOR              FROM             TO
AT            ON               IN               ?               WHO
WHAT         WHERE            HOW              DID              EAT
LIKE         THOUGHT          WAS              BUY              TAKE
TRAVEL
))

```

```

(setq letters '(
(JOHN          . (J O H N _ _ _ _ _)) (MARY          . (M A R Y _ _ _ _ _))

```

```

(MAMAISON . (M A M A I S O N _ _)) (LEONE'S . (L E O N E \ ' S _ _ _))
(LOBSTER . (L O B S T E R _ _ _)) (STEAK . (S T E A K _ _ _ _))
(DENNY'S . (D E N N Y \ ' S _ _ _)) (NORMS . (N O R M S _ _ _ _))
(SPAGHETTI . (S P A G H E T T I _)) (FISH . (F I S H _ _ _ _))
(MCDONALD'S . (M C D O N A L D \ ' S)) (BURGERKING . (B U R G E R _ K I N G))
(HAMBURGER . (H A M B U R G E R _)) (FRIES . (F R I E S _ _ _ _))
(BROADWAY . (B R O A D W A Y _ _)) (BULLOCK'S . (B U L L O C K \ ' S _ _))
(SHOES . (S H O E S _ _ _)) (JEANS . (J E A N S _ _ _ _))
(RADIO SHACK . (R A D I O S H A C K)) (CIRCUITCTY . (C I R C U I T C T Y))
(CD-PLAYER . (C D _ P L A Y E R _)) (TV . (T V _ _ _ _))
(RALPH'S . (R A L P H \ ' S _ _)) (SAFEGWAY . (S A F E W A Y _ _ _))
(SOFTDRINKS . (S O F T D R I N K S)) (VEGETABLES . (V E G E T A B L E S))
(LAX . (L A X _ _ _ _)) (DFW . (D F W _ _ _ _))
(JFK . (J F K _ _ _ _)) (SFO . (S F O _ _ _ _))
(CENTRALSTA . (C E N T R A L S T A)) (DOWNTOWNST . (D O W N T O W N S T))
(NEWYORK . (N E W Y O R K _ _)) (BOSTON . (B O S T O N _ _ _))
(BUS-TERMIN . (B U S - T E R M I N)) (BUS-STOP . (B U S - S T O P _ _))
(DOWNTOWN . (D O W N T O W N _ _)) (BEACH . (B E A C H _ _ _))
(WAITER . (W A I T E R _ _ _)) (CASHIER . (C A S H I E R _ _ _))
(CONDUCTOR . (C O N D U C T O R _)) (STAFF . (S T A F F _ _ _))
(DRIVER . (D R I V E R _ _ _)) (WENT . (W E N T _ _ _ _))
(SEATED . (S E A T E D _ _ _)) (ASKED . (A S K E D _ _ _))
(WAILED . (W A I T E D _ _ _)) (ATE . (A T E _ _ _ _))
(TASTED . (T A S T E D _ _ _)) (PAID . (P A I D _ _ _ _))
(LEFT . (L E F T _ _ _ _)) (PUNCHED . (P U N C H E D _ _ _))
(TRAVELED . (T R A V E L E D _ _)) (LOOKED . (L O O K E D _ _ _))
(TRIED . (T R I E D _ _ _ _)) (COMPARED . (C O M P A R E D _ _ _))
(TOOK . (T O O K _ _ _ _)) (CHECKED-IN . (C H E C K E D - I N))
(TOOK-OFF . (T O O K - O F F _ _)) (GOT-ON . (G O T - O N _ _ _))
(ARRIVED . (A R R I V E D _ _)) (GOT-OFF . (G O T - O F F _ _ _))
(BOUGHT . (B O U G H T _ _ _)) (TIP . (T I P _ _ _ _))
(NONE . (N O N E _ _ _ _)) (LINE . (L I N E _ _ _ _))
(SEVERAL . (S E V E R A L _ _ _)) (PRICES . (P R I C E S _ _ _))
(GOOD . (G O O D _ _ _ _)) (BAD . (B A D _ _ _ _))
(BIG . (B I G _ _ _ _)) (SMALL . (S M A L L _ _ _ _))
(BEST . (B E S T _ _ _ _)) (QUESTIONS . (Q U E S T I O N S _ _))
(CART . (C A R T _ _ _ _)) (GATE . (G A T E _ _ _ _))
(FLIGHT . (F L I G H T _ _ _)) (BOARDING . (B O A R D I N G _ _))
(TICKET . (T I C K E T _ _ _)) (PLANE . (P L A N E _ _ _))
(TRAIN . (T R A I N _ _ _)) (BUS . (B U S _ _ _ _))
(DISTANCE . (D I S T A N C E _ _)) (\ . (\ _ _ _ _))
(A . (A _ _ _ _)) (THE . (T H E _ _ _ _))
(ABOUT . (A B O U T _ _ _)) (FOR . (F O R _ _ _ _))
(FROM . (F R O M _ _ _)) (TO . (T O _ _ _ _))
(AT . (A T _ _ _ _)) (ON . (O N _ _ _ _))
(IN . (I N _ _ _ _)) (?) . (? _ _ _ _))
(WHO . (W H O _ _ _ _)) (WHAT . (W H A T _ _ _ _))
(WHERE . (W H E R E _ _ _)) (HOW . (H O W _ _ _ _))
(DID . (D I D _ _ _ _)) (EAT . (E A T _ _ _ _))
(LIKE . (L I K E _ _ _)) (THOUGHT . (T H O U G H T _ _ _))
(WAS . (W A S _ _ _)) (BUY . (B U Y _ _ _))
(TAKE . (T A K E _ _ _)) (TRAVEL . (T R A V E L _ _ _))
))

```

```

(setq values
 '( ( _ . 0.0) (A . 0.55) (B . 0.95) (C . 0.55) (D . 0.75) (E . 0.85)
 (F . 0.55) (G . 0.9) (H . 0.8) (I . 0.3) (J . 0.4) (K . 0.6)
 (L . 0.4) (M . 1.0) (N . 0.8) (O . 0.75) (P . 0.7) (Q . 0.8)
 (R . 0.9) (S . 0.65) (T . 0.45) (U . 0.65) (V . 0.5) (W . 0.95)
 (X . 0.55) (Y . 0.35) (Z . 0.65) (\ . 0.15) (? . 0.4) (- . 0.25)
 (\' . 0.2)
 ))

```

```

(defun explode (word)
  (cdr (assoc word letters)))

(defun convert (word)
  (mapcar 'getvalue (explode word)))

(defun getvalue (letter)
  (cdr (assoc letter values)))

```

```
(defun spit-out (allwords)
  (dolist (word allwords)
    (dolist (letter word)
      (format t "~&F" letter))))

(defun gener ()
  (spit-out (mapcar 'convert voc)))
```

The input data consists of pairs of symbols and concepts. The mapping between them is one-to-one in the DISCERN data, and a simple LISP program generates the training set.

```
(setq lwords
  '(( ( _ . -1)
      ( JOHN . 0)
      ( MARY . 1)
      ( MAMAISON . 2)
      ;;etc
      ( TRAVEL . 105)
      ))

(setq swords lwords)

(setq pairs
  '(( ( JOHN . John )
      ( MARY . Mary )
      ( MAMAISON . MaMaison )
      ;;etc
      ( TRAVEL . travel )
      ))

(defun gener ()
  (dolist (pair pairs)
    (format t "~&S ~S"
      (cdr (assoc (car pair) lwords))
      (cdr (assoc (cdr pair) swords)))))
```

C.3.3 Training data

The simulation file for the lexicon has the same basic format as the simulation files for the FGREP modules and the hierarchical feature maps. The file originally specifies representation and input file names and the simulation parameters, and snapshots are appended to the end of the file. Below, the top of the final snapshot (at the 500th epoch) contains the weights of the lexical unit (0,0) (top left corner of figure 9.7), which stand for the lexical representation of LINE. The next 12 weights shown belong to the semantic unit (8,1), which represents the semantic word line. The last 8 numbers are weights on the associative connections between lexical unit (0,0) and semantic units (8,0) to (8,3). The connections between LINE and line are almost 1, and there are also strong connections from the neighboring semantic units to LINE.

```
lreps          ;;; lexical representations
sreps75        ;;; semantic representations
inp            ;;; inputfile
0 0 100 500    ;;; displaying,testing,seed, tend
20 0 100 500 14 1 0.1 0.05 ;;; lnets,lt0,lt1,lt2,lnct0,lnct1,lalphat0,lalphat1
20 0 100 500 14 1 0.1 0.05 ;;; snets,st0,st1,st2,snct0,snct1,salphat0,salphat1
  0 100 500 0 0 0.1 0.05    ;;; at0,at1,at2,foo,foo,aalphat0,aalphat1
500 999999999  ;;; snapshots (epoch):-1=initial, 9x9=last
```



```

500
0.406250 0.281250 0.718749 0.749999 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000
...
0.399700 0.187871 0.208139 0.000001 0.281489 0.327466 0.705809 0.488658
0.967082 0.382052 0.940291 0.961299
...
0.035084   ;; lexical unit (0,0) to semantic unit (8,0)
0.757763   ;; semantic unit (8,0) to lexical unit (0,0)
0.996445   ;; lexical unit (0,0) to semantic unit (8,1)
0.998016   ;; semantic unit (8,1) to lexical unit (0,0)
0.035084   ;; lexical unit (0,0) to semantic unit (8,2)
0.468648   ;; semantic unit (8,2) to lexical unit (0,0)
0.000000   ;; lexical unit (0,0) to semantic unit (8,3)
0.000000   ;; semantic unit (8,3) to lexical unit (0,0)
...

```

As discussed in section 9.4, the lexicon is trained with the lexical and semantic vocabulary of the performance phase of DISCERN, i.e. with the word instances rather than the prototypes. Only words that appear in the input or output are included, i.e. representations for the script and track names (\$restaurant, \$fancy etc) are not used.

The training data (file `inp`) consists of lexical symbol – semantic concept pairs. If the data has ambiguities, there is one pair for each synonymous and homonymous sense of each word in the input file, e.g. CHICKEN -- prey and CHICKEN -- food (see appendix E.4.2). The DISCERN training data contains no ambiguities, however, and there is one-to-one mapping between symbols and concepts. Some words (e.g. the lexical ? and A) are repeated to make sure that they get separated on the map. A few samples from the input file are listed below. Again, the actual file contains indices instead of words. Index -1 indicates that the word is not presented to the corresponding map, and associative connections are not changed.

```

JOHN John
MARY Mary
MAMAISON Mamaison
LEONE'S Leone's
LOBSTER lobster
STEAK steak
...
A a
A -1
? ?
? -1
...

```

The file `lreps` contains the lexical representations of words. Below is the top of the file and representation components for JOHN and MARY:

```

lvoc           ;; lexical word labels (used for display only)
10             ;; lnrep
0.40625 0.6875 0.71875 0.71875 0.0 0.0 0.0 0.0 0.0 0.0
1.0      0.53125 0.78125 0.375  0.0 0.0 0.0 0.0 0.0 0.0
...

```

The file `sreps` contains the semantic representations, developed by the FGREP modules. The word prototypes have been replaced by the instance representations. Below, the first two vectors represent John and Mary, which share the same content part but have different IDs.

```
svoc          ;; semantic word labels (used for display only)
12           ;; snrep
0.750000 0.250000 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
0.250000 0.750000 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
...
```

APPENDIX D

DISCERN performance code and data

The core code for the connected DISCERN system is listed below. The program is divided into definitions, main control, FGREP modules (discussed in chapter 5), lexicon (chapter 9), hierarchical feature maps (chapter 7) and trace feature maps (chapter 8). Graphics routines, initializations and file I/O have been omitted from the listing. Before running DISCERN, the modules have been trained and the final weight values for each module have been stored in separate files (see section D.3 below).

The DISCERN program reads the weight data files and builds the complete model. It then reads story data and question answering data from user-specified input data files. The story data was created by a lisp-program, listed in section D.2. Samples from the performance test files and the input file for the example (which was presented in section 10.2) are listed in section D.3.

The DISCERN program consists of selected parts of the training programs (appendix C). Compared to the training code, the control for repeated input presentations, the weight modification routines, and the graphics calls are absent and the different networks have been brought together into a single program. In addition, this program generates textual log of its execution.

D.1 The DISCERN program

D.1.1 Definitions

```
#include <stdio.h>
#include <math.h>

/***** general *****/

#define maxmodules 12      /* 6 qanets + store + ret + l + s + ls + sl */
#define maxrep 12         /* maximum size of lexical & semantic reps */
#define maxwords 300      /* maximum number of lexical & semantic words*/
#define maxwordl 20       /* maximum length of word labels (chars) */
#define maxsent 7         /* max number of sentences in a story */
#define maxword 8         /* max number of words in a sentence */
#define maxcase 6         /* max number of case roles */
#define maxslot 7         /* max number of slots in storyrep */
#define maxas (maxcase+maxslot) /* max number of assemblies in any layer */
#define maxslotrep (maxslot*maxrep) /* max number of units in the storyrep */
#define maxsnaps 50       /* max number of snapshots */
#define maxinpstories 100 /* max number of stories in epoch */
#define maxtask 2         /* max number of tasks (para, qa..) */

#define paratask 0        /* index to task table */
#define qatask 1         /* index to task table */
#define sentgenmod 3     /* module # for sentence generator */
```

```

#define storemod 6          /* module # for storing into episodic mem */
#define retmod 7           /* module # for retrieving from episodic mem */
#define linpmod 8         /* module # for lexical input map */
#define soutmod 9         /* module # for semantic output map */
#define sinpmod 10        /* module # for semantic input map */
#define loutmod 11        /* module # for lexical output map */
#define firsttokword 108  /* the index of the first token word */
#define withinerr 0.15    /* statistics collected within this error */
#define discernprompt "DISCERN>"
#define snapshotend 99999999 /* token that indicates last snapshot in file*/

int nlwords, nlrep, nswords, nsrep, nstories, nquest, nmodules;
int chained, withlex, withhfm, context_mode, babbling, print_mistakes, task;

/* words */
struct wordstruct
{
    char chars[maxwordl]; /* label */
    float rep[maxrep];    /* representation */
};

/* lexical words */
char blanklchars[maxwordl];
float blanklrep[maxrep]; /* blanklrep=lwords[-1].rep */
struct wordstruct lwords[maxwords];

/* semantic words */
char blankschars[maxwordl];
float blanksrep[maxrep]; /* blanksrep=swords[-1].rep */
struct wordstruct swords[maxwords];

/* sentence data */
struct sentstruct
{
    int words[maxword], caserole[smaxcase], included;
};

/* story data */
struct
{
    int slots[maxslot]; /* slot-filler rep of the story */
    struct sentstruct sents[smaxsent]; /* sentences */
} story;

/* question and answer data */
struct
{
    int slots[maxslot]; /* slot-filler rep of the story */
    struct sentstruct question, answer; /* question and answer sentences */
} qa;

/* active reps on the pathways */
float swordrep[maxrep], caserep[smaxcase*maxrep], slotrep[smaxslotrep];
int nslotrep, ncaserep;

/* statistics */
int within[martask][maxmodules],
    blankall, all[martask][maxmodules][maxwords+1],
    blankcorr, corr[martask][maxmodules][maxwords+1];
float deltasum[martask][maxmodules];

/* files */
char initfile[100], inpfile[100], lrepfile[100], lvocfile[100],
    sreplefile[100], svocfile[100], qafile[100], hfmfile[100],
    tracefile[100], lexfile[100];
FILE *fp, *fp2, *infp;

/***** fgrep networks *****/

```

```

#define maxqanets 6          /* max number of fgrep networks in the system */
#define maxqaunits (maxas*maxrep)
#define maxhidrep 100      /* maximum size of the representations */
#define hidrepstep 5       /* steps for optimizing inner fwd prop loops: */
#define inpunitstep 4      /* must divide the number of hidden/input units */

/* actual numbers of assemblies */
int qanets, nslot, nword, ncase, nsent, nquest;

/* actual table dimensions */
int nhidrep[maxqanets], ninpunits[maxqanets], noutunits[maxqanets];

/* units and weights */
float inprep[maxqanets][maxqaunits], outrep[maxqanets][maxqaunits],
  tchrep[maxqanets][maxqaunits],
  prevhidrep[maxqanets][maxhidrep], hidrep[maxqanets][maxhidrep],
  hidbias[maxqanets][maxhidrep], outbias[maxqanets][maxqaunits],
  wih[maxqanets][maxqaunits][maxhidrep],
  who[maxqanets][maxqaunits][maxhidrep],
  wph[maxqanets][maxhidrep][maxhidrep];

/***** lexicon *****/

#define maxlsnets 20      /* max size of the map is 20x20 */
#define anc 1              /* neighborhood size */

int lnets, snets,         /* actual network sizes */
  abesti, abestj;        /* indices for the best associative unit */

/* units on the feature maps */
struct lexunit
{
  float value;           /* output value */
  float comp[maxrep];    /* input weight components */
} lunits[maxlsnets][maxlsnets], sunits[maxlsnets][maxlsnets];

/* associative connections */
float lassoc[maxlsnets][maxlsnets][maxlsnets][maxlsnets],
  slassoc[maxlsnets][maxlsnets][maxlsnets][maxlsnets];

/***** hfm *****/

#define maxhfmnets 2     /* maximum topnetwork size is 2*2 units */
#define maxmnets 2        /* maximum midnetwork size is 2*2 units */
#define maxbnets 8        /* maximum bottomnetwork size is 8*8 units */
#define hfmepsilon 2.2    /* look for trace under units within hfmeps */
#define mepsilon 3.0      /* look for trace under midunits within meps */

/* actual network sizes and indices for the image units */
int hfmnets, mnets, bnets, besti, bestj,
  tbesti, tbestj, mbesti, mbestj, bbesti, bbestj;

float inpvector[maxslotrep], outvector[maxslotrep];

/* compression */
int indices[maxhfmnets][maxhfmnets][maxslotrep],
  nmlines[maxhfmnets][maxhfmnets],
  mindices[maxhfmnets][maxhfmnets][maxmnets][maxmnets][maxslotrep],
  nblines[maxhfmnets][maxhfmnets][maxmnets][maxmnets];

/* units at the top, middle and bottom levels */
struct hfmunitdata
{
  float value;
  float comp[maxslotrep];
} hfmunits[maxhfmnets][maxhfmnets];

struct munitdata {
  float value;

```

```

float comp[maxslotrep];
} munits[maxhfmnets][maxhfmnets][maxmnets][maxmnets];

struct bunitdata {
float value; /* current output value */
float prevvalue; /* output during previous time step */
float comp[maxslotrep]; /* input weights */
float latweights[maxbnets][maxbnets]; /* lateral weights */
} bunits[maxhfmnets][maxhfmnets][maxmnets][maxmnets][maxbnets][maxbnets];

/***** trace *****/

#define epsilon 0.0001 /* test within epsilon */
#define aliveact 0.5 /* no trace found if max activity <aliveact */

int tracenc, tsettle; /* neighborhood size, settling iterations */
float minact, maxact, gammaexc, gammainh, /*sigmoid and lateral weight params*/
bestvalue; /* activity of maximally responding unit */

```

D.1.2 Main control

```

#include "discrndefs.h"

main(argc,argv)
int argc; char *argv[];
{
char s[100];
int i;
sprintf(initfile, "%s", argv[1]);
setbuf(stdout, NULL); /* log output one word at a time */
init_nets(); /* read in all networks (not listed) */

printwordout(discernprompt);
while(scanf("%s", inpf) != EOF)
{
if (inpf[0] == '-')
clear_traces(); /* clear the memory */
else
{
if(init_inp()) /* read the input file specs (not listed) */
{
init_stats(); /* clear statistics variables (not listed) */
while(fscanf(infp, "%d %d", &nstories, &nquest) != EOF)
{
fgets(s, 99, infp);
process_stories();
}
fclose(infp);
print_stats();
}
}
printwordout(discernprompt);
}
exit(0);
}

process_stories()
/* read and paraphrase stories and read and answer questions */
{
char s[100];
int i, j;

task=paratask;

```

```

if(babbling && nstories>0)
    printf("\n***** Read %d %s%s*****",
           nstories, (withhfm)?"and store ":"",
           (context_mode)?"and paraphrase ":"");
if(context_mode && withhfm) clear_traces();
for(i=0; i<nstories; i++)
    {
    read_story_data();           /* read story rep and the sentences */
    parse_story();             /* see section D.1.3 */
    if(withhfm)                /* if episodic memory included */
        presentmem(storemod,story.slots); /*store into memory; section D.1.5 */
    if(context_mode)          /* paraphrase in context mode only */
        gener_story();         /* see section D.1.3 */
    }

task=qatask;
if(babbling && nquest>0)
    printf("\n***** Read %d questions %sand answer %s*****",
           nquest, (withhfm)?"and retrieve ":"",
           (context_mode)?"in context ":"");
if(context_mode)
    for(i=0; i<nslot; i++)      /* in context mode, */
        qa.slots[i]=story.slots[i]; /* all questions refer to the same story */
for(i=0; i<nquest; i++)
    {
    if(!context_mode)
        {
        for(j=0; j<nslot; j++)
            fscanf(inpfp, "%d",&qa.slots[j]); /* read the target story */
            fgets(s,99,inpfp);
        }
    read_qa_data();           /* read question and answer sentences */
    if(qa.question.included>0)
        {
        formcue();           /* see section D.1.3 */
        if(withhfm)
            presentmem(retmod,qa.slots); /*retrieve from memory;section D.1.5*/
        }
    if(!(qa.answer.included<=0 || withhfm && bestvalue<aliveact))
        produceanswer();     /* see section D.1.3 */
    }
}

/***** math routines *****/

float distance(v1, v2, nrep)
/* compute distance of two nrep-dimensional vectors */
float v1[], v2[];
int nrep;
{
    float sum=0.0;
    register int i;
    for(i=0; i<nrep; i++)
        sum += (v1[i]-v2[i])*(v1[i]-v2[i]);
    return(sqrt(sum));
}

float seldistance(indices, v1, v2, ncomp)
/* compute distance of two ncomp-dimensional vectors
   obtained by selecting components from v1 and v2 */
int ncomp, indices[];
float v1[],v2[];
{
    float sum=0.0;
    register int i;
    /* count only selected components in the distance */

```

```

    for(i=0; i<ncomp; i++)
        sum=sum+(v1[indices[i]]-v2[i])*(v1[indices[i]]-v2[i]);
    return(sqrt(sum));
}

/***** stats routines *****/

collect_stats(modi, outrep, nas, target, words, nrep, nwords)
struct wordstruct words[];
float outrep[];
int modi, nas, target[], nrep, nwords;
{
    int i, nearest, j;
    float uniterror;
    char s[100];

    for(i=0; i<nas; i++)
        for(j=0; j<nrep; j++)
        {
            uniterror = fabs(words[target[i]].rep[j] - outrep[i*nrep+j]);
            deltasum[task][modi] += uniterror; /* cumulate total error */
            if (uniterror < withinerr)
                within[task][modi]++; /* cumulate # of units within 0.15*/
        }

    /* cumulate # of correct words */
    for(i=0; i<nas; i++)
    {
        all[task][modi][ target[i] ]++;
        nearest = find_nearest_s(&outrep[i*nrep], words, nrep, nwords);
        if(nearest == target[i])
        {
            corr[task][modi][ target[i] ]++;
            if(babbling) printwordout(words[nearest].chars);
        }
        else if(print_mistakes || babbling)
        {
            sprintf(s, "%d:%s(%s)",
                    modi, words[nearest].chars, words[target[i]].chars);
            printwordout(s);
            if(!babbling) printf("\n");
        }
    }
    if(babbling && modi!=sentgenmod && modi<linpmod) printf("\n");
}

print_stats()
{
    print_task_stats("paraphrasing", paratask);
    print_task_stats("question answering", qatask);
}

print_task_stats(taskname, taski)
int taski;
char taskname[];
{
    int modi, nwords, nrep, i, j, sum_corr, sum_all, sum_corrtok, sum_alltok;

    printf("\n\nAverage %s performance:\n", taskname);
    printf("Module    All    Syn    <%.2f    Err:\n", withinerr);
    for(modi=0; modi<nmodules; modi++)
    {
        nwords=(modi==linpmod || modi==loutmod)?nlwords:nwords;
        nrep=(modi==linpmod || modi==loutmod)?nlrep:nrep;
        sum_corr = sum_all = sum_corrtok = sum_alltok = 0;
        /* count the total of correct and all words */
        for(i=-1; i<nwords; i++)
        {
            sum_corr += corr[taski][modi][i];

```



```

    sum_all += all[taski][modi][i];
}
/* count the total of correct and all words for instance words */
for(i=firsttokword; i<nwords; i++)
{
    sum_corrtok += corr[taski][modi][i];
    sum_alltok += all[taski][modi][i];
}
if(sum_alltok>0)
{
    printf("Module %d: ", modi);
    printf("%6.1f %6.1f %6.1f %9.4f",
        100.0 * sum_corr / sum_all,
        100.0 * sum_corrtok / sum_alltok,
        100.0 * within[taski][modi] / (sum_all * nrep),
        deltasum[taski][modi] / (sum_all * nrep));
    printf("  %d %d %d %d %d %f\n", sum_all, sum_corr, sum_alltok,
        sum_corrtok, within[taski][modi], deltasum[taski][modi]);
}
}
}

int find_nearest_s(rep, words, nrep, nwords)
/* find the representation that is nearest to rep in the lexicon */
struct wordstruct words[];
float rep[];
int nrep,nwords;
{
    int i,bestindex;
    float lbest, dist, distance();
    lbest=999999999.9;
    for(i=(-1); i<nwords; i++)
    {
        dist= distance(rep,words[i].rep,nrep);
        if(dist<lbest)
        {
            bestindex=i;
            lbest=dist;
        }
    }
    return(bestindex);
}

```

D.1.3 FGREP modules

```

#include "discerndefs.h"

/***** FGREP modules *****/

parse_story()
/* story parser module: sequential input FGREP */
{
    register int i,j,senti;

    if(babbling) printf("\n\n[ parsing input story: ]\n");

    for(i=0; i<nhidrep[1]; i++)
        prevhidrep[1][i]=0;
    /* read all sentences which are included in the story */
    for(senti=0; senti<nsent; senti++)
        if(story.sents[senti].included>0)
        {
            parse_sentence(story.sents[senti]);
            if (chained)
                for(i=0; i<ninpunits[1]; i++)
                    inprep[1][i] = caserep[i]; /* use active sentence */
        }
}

```

```

        else
            for(i=0; i<ncase; i++)
                for(j=0; j<nsrep; j++)
                    inprep[1][i*nsrep+j] = /* use correct input patterns */
                        swords[story.sents[senti].caseroles[i]].rep[j];
            propagate_and_display(1);
        }
    if(babbling) printf("\n[ into internal rep: ]\n");
    collect_stats(1, outrep[1], nslot, story.slots, swords, nsrep, nswords);

    if (chained)
        for (i=0; i<nslotrep; i++)
            slotrep[i] = outrep[1][i]; /* activate result */
}

parse_sentence(sent)
/* sentence parser module: sequential input FGREP */
struct sentstruct sent;
{
    int i,j,wordi;

    for(i=0; i<nhidrep[0]; i++)
        prevhidrep[0][i]=0;
    /* read all words of the sentence */
    for(wordi=0; wordi<nword; wordi++)
        if(sent.words[wordi]>(-1)) /* -1 = filler for short sentences */
            {
                if (withlex) /* lexicon model included, use it */
                    input_lexicon(sent.words[wordi]);
                else
                    if(babbling) printwordout(swords[sent.words[wordi]].chars);
                    if (chained && withlex)
                        for(i=0; i<nsrep; i++)
                            inprep[0][i] = swordrep[i]; /* use active concept rep */
                    else
                        for(i=0; i<nsrep; i++) /* use correct input patterns */
                            inprep[0][i] = swords[sent.words[wordi]].rep[i];
                    propagate_and_display(0);
            }
    if(babbling) printf("\n");
    collect_stats(0, outrep[0], ncase, sent.caseroles, swords, nsrep, nswords);

    if(chained)
        for (i=0; i<ncaserep; i++)
            caserep[i] = outrep[0][i]; /* activate result */
}

gener_story()
/* story generator module: sequential output FGREP */
{
    register int i,j,senti;

    if(babbling) printf("\n[ generating paraphrase: ]\n");
    for (i=0; i<nslot; i++)
        for(j=0; j<nsrep;j++)
            if(chained)
                inprep[2][i*nsrep+j] = slotrep[i*nsrep+j]; /* use active story */
            else
                /* use correct representations */
                inprep[2][i*nsrep+j] = swords[story.slots[i]].rep[j];

    for(i=0; i<nhidrep[2]; i++)
        prevhidrep[2][i]=0;
    /* generate each sentence */
    for(senti=0; senti<nsent; senti++)
        if(story.sents[senti].included>(-1))
            {
                propagate_and_display(2);
            }
}

```

```

        if (chained)
            for (i=0; i<ncaserep; i++)
                caserep[i] = outrep[2][i]; /* activate result */

        collect_stats(2, outrep[2], ncase, story.sents[senti].caseroles,
                    swords, nsrep, nswords);
        gener_sentence(story.sents[senti]);
    }
}

```

```

gener_sentence(sent)
/* sentence generator module: sequential output FGREP */
struct sentstruct sent;
{
    register int i,j,wordi;

    for (i=0; i<ncase; i++)
        for(j=0; j<nsrep;j++)
            if(chained)
                inprep[3][i*nsrep+j] = caserep[i*nsrep+j]; /* use active sent */
            else
                /* use correct representations */
                inprep[3][i*nsrep+j] = swords[sent.caseroles[i]].rep[j];

    for(i=0; i<nhidrep[3]; i++)
        prevhidrep[3][i]=0;
    /* generate the words one at a time */
    for(wordi=0; wordi<nword; wordi++)
        if(sent.words[wordi]>(-1))
            {
                propagate_and_display(3);
                collect_stats(3, outrep[3], 1, &sent.words[wordi],
                            swords, nsrep, nswords);
                if(chained && withlex)
                    for(j=0; j<nsrep; j++)
                        swordrep[j]=outrep[3][j]; /* activate result */
                if (withlex) /* if lexicon is included */
                    output_lexicon(sent.words[wordi]);
            }
    if(babbling) printf("\n");
}

```

```

formcue()
/* cue former module: nonrecurrent FGREP */
{
    register int i,j;

    if(babbling) printf("\n[ parsing question: ]\n");
    parse_sentence(qa.question);

    if(chained)
        for (i=0; i<ninpunits[4]; i++)
            inprep[4][i]=caserep[i]; /* use active sent */
        else
            /* use correct representations */
            for (i=0; i<ncase; i++)
                for(j=0; j<nsrep;j++)
                    inprep[4][i*nsrep+j] =
                        swords[ qa.question.caseroles[i] ].rep[j];

    propagate_and_display(4);
    if(babbling) printf("\n[ into cue: ]\n");
    collect_stats(4, outrep[4], nslot, qa.slots, swords, nsrep, nswords);

    if(chained && withhfm)
        for(i=0; i<nslotrep; i++)
            slotrep[i]=outrep[4][i]; /* activate result */
}

```

```

}
produceanswer()
/* answer producer module: nonrecurrent FGREP */
{
  register int i,j;
  if(chained)
  {
    for (i=0; i<noutunits[0]; i++)
      inprep[5][i]=caserep[i]; /* use active sent */
    for (i=0; i<noutunits[1]; i++)
      inprep[5][i+noutunits[0]] = slotrep[i]; /* use active slots */
  }
  else /* use correct representations */
  {
    for (i=0; i<ncase; i++)
      for(j=0; j<nsrep;j++)
        inprep[5][i*nsrep+j] =
          swords[ qa.question.caseroles[i] ].rep[j];
    for (i=0; i<nslot; i++)
      for(j=0; j<nsrep;j++)
        inprep[5][noutunits[0]+i*nsrep+j] =
          swords[ qa.slots[i] ].rep[j];
  }
  propagate_and_display(5);
  if(babbling) printf("\n[ generating answer: ]\n");
  collect_stats(5, outrep[5], ncase, qa.answer.caseroles,
               swords, nsrep, nswords);

  if(chained)
    for(i=0; i<ncaserep; i++)
      caserep[i] = outrep[5][i]; /* activate result */
  gener_sentence(qa.answer);
}

/***** propagation *****/
propagate_and_display(neti)
/* calls to display routines are not listed, sorry */
int neti;
{
  register int i,k;
  forward_prop(neti);
  if (neti<4) /* recurrent FGREP modules */
    for(k=0; k<nhidrep[neti]; k++) /* cp hidden layer to previous h.l. */
      prevhidrep[neti][k]=hidrep[neti][k];
}

forward_prop(neti)
/* forward propagation in the FGREP modules;
   inner loops have been rolled out for more efficient computation */
register int neti;
{
  register int i,j,k,p;

  /* set up bias for the hidden units */
  for(k=0; k<nhidrep[neti]; k++)
    hidrep[neti][k]=hidbias[neti][k];
  /* propagate from the previous hidden layer to h.l. (recurrent FGREP only) */
  if (neti<4)
    for(p=hidrepstep-1; p<nhidrep[neti]; p+=hidrepstep)
      for(k=0; k<nhidrep[neti]; k++)
        hidrep[neti][k] += prevhidrep[neti][p]*wph[neti][p][k]
          + prevhidrep[neti][p-1]*wph[neti][p-1][k]

```

```

        + prevhidrep[neti][p-2]*wph[neti][p-2][k]
        + prevhidrep[neti][p-3]*wph[neti][p-3][k]
        + prevhidrep[neti][p-4]*wph[neti][p-4][k];
/* propagate from the input layer to the hidden layer */
for(i=inpunitstep-1; i<ninpunits[neti]; i+=inpunitstep)
    for(k=0; k<nhidrep[neti]; k++)
        hidrep[neti][k] += inprep[neti][i]*wih[neti][i][k]
            + inprep[neti][i-1]*wih[neti][i-1][k]
            + inprep[neti][i-2]*wih[neti][i-2][k]
            + inprep[neti][i-3]*wih[neti][i-3][k];
/* sigmoid the output layer */
for(k=0; k<nhidrep[neti]; k++)
    hidrep[neti][k] = 1.0/(1.0+exp(-hidrep[neti][k]));

/* set up bias for the output units */
for(i=0; i<noutunits[neti]; i++)
    outrep[neti][i]=outbias[neti][i];
/* propagate from the hidden layer to the output layer */
for(k=hidrepstep-1; k<nhidrep[neti]; k+=hidrepstep)
    for(i=0; i<noutunits[neti]; i++)
        outrep[neti][i] += hidrep[neti][k]*who[neti][i][k]
            + hidrep[neti][k-1]*who[neti][i][k-1]
            + hidrep[neti][k-2]*who[neti][i][k-2]
            + hidrep[neti][k-3]*who[neti][i][k-3]
            + hidrep[neti][k-4]*who[neti][i][k-4];
/* sigmoid the output layer */
for(i=0; i<noutunits[neti]; i++)
    outrep[neti][i] = 1.0/(1.0+exp(-outrep[neti][i]));
}

```

D.1.4 Lexicon

```

#include "discerndefs.h"

input_lexicon(index)
/* translate a lexical rep into its semantic counterpart */
int index;
{
    int i;
    float inpwordrep[maxrep];
    char s[100];

    for(i=0; i<nlrep; i++)
        inpwordrep[i]=lwords[index].rep[i];          /* form input */
    translate(index, inpwordrep, linpmod, lnets, lunits, lwords, nlrep, nlwords,
        soutmod, snets, sunits, swords, nsrep, nswords, lsassoc);
    if(chained)
        for(i=0; i<nsrep; i++)
            swordrep[i] = sunits[abesti][abestj].comp[i];    /* activate result */
}

output_lexicon(index)
/* translate a semantic rep into its lexical counterpart */
int index;
{
    int i;
    float inpwordrep[maxrep];

    if(chained)
        for(i=0; i<nsrep; i++)
            inpwordrep[i]=swordrep[i];                /* use current sword */
    else
        for(i=0; i<nsrep; i++)
            inpwordrep[i]=swords[index].rep[i];        /* use correct reps */
}

```

```

    translate(index, inpwordrep, sinpmod, snets, sunits, swords, nsrep, nswords,
              loutmod, lnets, lunits, lwords, nlrep, nlwords, slassoc);
}

translate(index, inpwordrep, inpmod, inpnets, inpunits, inpwords, inpnrep, inpnwords,
          outmod, outnets, outunits, outwords, outarep, outnwords, assoc)
struct lexunit inpunits[maxlsnets][maxlsnets], outunits[maxlsnets][maxlsnets];
float inpwordrep[], assoc[maxlsnets][maxlsnets][maxlsnets][maxlsnets];
struct wordstruct inpwords[], outwords[];
int index, inpmod, inpnets, inpnrep, inpnwords, outmod, outnets, outnrep, outnwords;
{
    int i, j, ii, jj, besti, bestj;
    float distance();
    float best, second;
    int lowi, lowj, highi, highj, secondi, secondj, thirdi, thirdj, fourthi, fourthj;
    float highest, third, fourth;

    /* present the input vector to the input map and find the image unit */
    second=999999999.9;
    best=999999999.9;
    for(i=0; i<inpnets; i++)
        for(j=0; j<inpnets; j++)
            {
                inpunits[i][j].value= distance(inpwordrep, inpunits[i][j].comp, inpnrep);
                if (inpunits[i][j].value == best) second=best;
                if (inpunits[i][j].value < best)
                    {
                        besti=i; bestj=j; best=inpunits[i][j].value;
                    }
            }
    if (babbling && second == best)
        printf("\nWarning: input image for %s is not unique: %f\n",
              inpwords[index].chars, best);
    collect_stats(inpmod, inpunits[besti][bestj].comp, 1,
                 &index, inpwords, inpnrep, inpnwords);

    /* clear the output map */
    for(ii=0; ii<outnets; ii++)
        for(jj=0; jj<outnets; jj++)
            outunits[ii][jj].value=0.0;

    /* set up neighborhood */
    if (besti-anc>=0) lowi=besti-anc; else lowi=0;
    if (besti+anc<inpnets) highi=besti+anc; else highi=inpnets-1;
    if (bestj-anc>=0) lowj=bestj-anc; else lowj=0;
    if (bestj+anc<inpnets) highj=bestj+anc; else highj=inpnets-1;

    /* find the four most active units in the neighborhood */
    fourth=third=second=best=999999999.9;
    highest=(-1.0);
    for(i=lowi; i<=highi; i++)
        for(j=lowj; j<=highj; j++)
            {
                if (inpunits[i][j].value < best)
                    {
                        fourthi=thirdi; fourthj=thirdj; fourth=third;
                        thirdi=secondi; thirdj=secondj; third=second;
                        secondi=besti; secondj=bestj; second=best;
                        besti=i; bestj=j; best=inpunits[i][j].value;
                    }
                else if (inpunits[i][j].value < second)
                    {
                        fourthi=thirdi; fourthj=thirdj; fourth=third;
                        thirdi=secondi; thirdj=secondj; third=second;
                        secondi=i; secondj=j; second=inpunits[i][j].value;
                    }
                else if (inpunits[i][j].value < third)

```

```

        {
            fourthi=thirdi; fourthj=thirdj; fourth=third;
            thirdi=i; thirdj=j; third=inpunits[i][j].value;
        }
    else if (inpunits[i][j].value < fourth)
    {
        fourthi=i; fourthj=j; fourth=inpunits[i][j].value;
    }
    if(inpunits[i][j].value>highest)
        highest=inpunits[i][j].value;
}

/* propagate through these units to the output map;
   equations 9.1 and 9.3 (the sigmoid has been dropped because
   we are only interested in finding the max unit, not its actual value */
for(i=lowi; i<=highi; i++)
    for(j=lowj; j<=highj; j++)
        if((i==besti && j==bestj) ||
            (i==secondi && j==secondj) ||
            (i==thirdi && j==thirdj) ||
            (i==fourthi && j==fourthj))
            for(ii=0; ii<outnets; ii++)
                for(jj=0; jj<outnets; jj++)
                    outunits[ii][jj].value +=
                        (1.0-(inpunits[i][j].value-best)/(highest-best))
                        * assoc[i][j][ii][jj];

/* find the image unit in the output map */
second=(-1.0);
best=(-1.0);
for(ii=0; ii<outnets; ii++)
    for(jj=0; jj<outnets; jj++)
        {
            if(outunits[ii][jj].value==best) second=best;
            if(outunits[ii][jj].value>best)
                {
                    abesti=ii; abestj=jj; best=outunits[ii][jj].value;
                }
        }
if (babbling && second == best)
    printf("\nWarning: assoc image for %s is not unique: %f\n",
           outwords[index].chars, best);
collect_stats(outmod, outunits[abesti][abestj].comp, 1,
              &index, outwords, outnrep, outnwords);
}

```

D.1.5 Hierarchical feature maps

```

#include "discerndefs.h"

presentmem(mod,slots)
/* present an input vector to the episodic memory
   and either store it or use it as a cue to retrieve
   a previously stored vector */
int mod, slots[];
{
    int i;

    form_invector(slots);

    if(mod==storemod)
        classify_and_store();
    else if(mod==retmod)
        classify_and_retrieve();

    form_outvector(mod);
}

```

```

collect_stats(mod, outvector, nslot, slots, swords, nsrep, nswords);
if(chained)
    for(i=0; i<nslotrep; i++)
        slotrep[i]= outvector[i]; /* activate result */
}

classify_and_store()
/* storing */
{
    register int i,j;
    float distance(), seldistance();
    float best=999999999.9;

    if(babbling) printf("\n[ storing into episodic memory: ]\n");
    /* find the image unit at the script level */
    for(i=0; i<hfmnets; i++)
        for(j=0; j<hfmnets; j++)
            {
                hfmunits[i][j].value =
                    distance(inpvector,
                        hfmunits[i][j].comp,
                        nslotrep);
                /* check if this unit's response is best so far encountered */
                if (hfmunits[i][j].value < best)
                    {
                        tbesti=i; tbestj=j;
                        best=hfmunits[i][j].value;
                    }
            }

    /* find the image unit at the track level */
    best=999999999.9;
    for(i=0; i<mnets; i++)
        for(j=0; j<mnets; j++)
            {
                munits[tbesti][tbestj][i][j].value =
                    seldistance(indices[tbesti][tbestj], inpvector,
                        munits[tbesti][tbestj][i][j].comp,
                        nmlines[tbesti][tbestj]);
                if (munits[tbesti][tbestj][i][j].value < best)
                    {
                        mbesti=i; mbestj=j;
                        best=munits[tbesti][tbestj][i][j].value;
                    }
            }

    /* find the image unit at the role-binding level */
    best=999999999.9;
    for(i=0; i<bnets; i++)
        for(j=0; j<bnets; j++)
            {
                bunits[tbesti][tbestj][mbesti][mbestj][i][j].value =
                    seldistance(mindices[tbesti][tbestj][mbesti][mbestj], inpvector,
                        bunits[tbesti][tbestj][mbesti][mbestj][i][j].comp,
                        nblines[tbesti][tbestj][mbesti][mbestj]);
                if (bunits[tbesti][tbestj][mbesti][mbestj][i][j].value < best)
                    {
                        bbesti=i; bbestj=j;
                        best=bunits[tbesti][tbestj][mbesti][mbestj][i][j].value;
                    }
            }

    /* create a trace at the role-binding level */
    store_trace(bunits[tbesti][tbestj][mbesti][mbestj],
        nblines[tbesti][tbestj][mbesti][mbestj],
        mindices[tbesti][tbestj][mbesti][mbestj]);
}

```



```

}

classify_and_retrieve()
/* retrieving */
{
    int i,j,ii,jj,ci0,cj0,ci1,cj1,ci2,cj2;
    float distance(), seldistance();
    float bestcvalue=(-2.0);

    if(babbling) printf("\n[ retrieving from episodic memory: ]\n");
    for(i=0; i<hfmnets; i++)
        for(j=0; j<hfmnets; j++)
            {
                hfmunits[i][j].value= distance(inpvector,hfmunits[i][j].comp,nslotrep);
                printf("%d %d: %f\n",i,j,hfmunits[i][j].value);
            }

    for(i=0; i<hfmnets; i++)
        for(j=0; j<hfmnets; j++)
            /* look for the appropriate trace under all scripts
            better than hfmepsilon */
            if(hfmunits[i][j].value < hfmepsilon)
                {
                    for(ii=0; ii<mnets; ii++)
                        for(jj=0; jj<mnets; jj++)
                            {
                                munits[i][j][ii][jj].value =
                                    seldistance(indices[i][j], inpvector,
                                                munits[i][j][ii][jj].comp,
                                                nmlines[i][j]);
                                printf("%d %d, %d %d: %f\n",i,j,ii,jj,
                                        munits[i][j][ii][jj].value);
                            }
                    /* look for the appropriate trace under all tracks
                    better than mepsilon */
                    for(ii=0; ii<mnets; ii++)
                        for(jj=0; jj<mnets; jj++)
                            if(munits[i][j][ii][jj].value < mepsilon)
                                {
                                    if(babbling) printf("\n[ map %d %d, %d %d:", i, j, ii, jj);
                                    /* try to retrieve the trace from this role-binding map */
                                    retrieve_trace(bunits[i][j][ii][jj],
                                                    nblines[i][j][ii][jj],
                                                    mindices[i][j][ii][jj]);
                                    if(bestvalue>bestcvalue)
                                        {
                                            ci0=i; cj0=j;
                                            ci1=ii; cj1=jj;
                                            ci2=bbesti; cj2=bbestj;
                                            bestcvalue=bestvalue;
                                        }
                                }
                }

    /* this is the best trace that was found in the memory */
    tbesti=ci0; tbestj=cj0;
    mbesti=ci1; mbestj=cj1;
    bbesti=ci2; bbestj=cj2;
    bestvalue=bestcvalue;
}

form_inpvector(slots)
int slots[];
{
    register int i,j;

    if(chained)
        for(i=0; i<nslotrep; i++)
            inpvector[i] = slotrep[i];          /* use current story */
}

```

```

else
    for(i=0; i<nslot; i++)
        for(j=0; j<nsrep; j++)
            inpvector[i*nsrep+j] = swords[slots[i]].rep[j];
}

form_outvector(mod)
/* decide whether a trace was found and form the retrieved vector */
int mod;
{
    int i;

    if(mod==retmod && bestvalue<aliveact)
    {
        if(babbling || print_mistakes)
            printf("[ oops: no image found ]\n");
        for(i=0; i<nslotrep; i++)
            outvector[i] = 0.0;
    }
    else
    {
        /* a trace was retrieved */
        if(babbling) printf("\n[ image units (%d,%d), (%d,%d), (%d,%d): ]\n",
            tbesti, tbestj, mbesti, mbestj, bbesti, bbestj);
        /* combine the weights of the image units */
        for(i=0; i<nslotrep; i++)
            outvector[i] = hfmunits[tbesti][tbestj].comp[i];
        for (i=0; i<nmlines[tbesti][tbestj]; i++)
            outvector[ indices[tbesti][tbestj][i] ] =
                munits[tbesti][tbestj][mbesti][mbestj].comp[i];
        for (i=0; i<nblines[tbesti][tbestj][mbesti][mbestj]; i++)
            outvector[ mindices[tbesti][tbestj][mbesti][mbestj][i] ] =
                bunits[tbesti][tbestj][mbesti][mbestj][bbesti][bbestj].comp[i];
    }
}

```

D.1.6 Trace feature maps

```

#include "discerndefs.h"

store_trace(table, dim, indices)
struct bunitdata table[maxbnets][maxbnets];
int dim, indices[];
{
    register int i,j, ii, jj;
    int lowi, lowj, highi, highj, foobesti2, foobestj2;
    float lowest, highest;

    /* move the neighborhood away from the boundaries */
    if (besti2<tracenc) foobesti2=tracenc;
    else if(besti2>bnets-1-tracenc) foobesti2=bnets-1-tracenc;
    else foobesti2=besti2;
    if (bestj2<tracenc) foobestj2=tracenc;
    else if(bestj2>bnets-1-tracenc) foobestj2=bnets-1-tracenc;
    else foobestj2=bestj2;

    /* set up neighborhood limits */
    lowi=foobesti2-tracenc;
    highi=foobesti2+tracenc;
    lowj=foobestj2-tracenc;
    highj=foobestj2+tracenc;

    /* find the most and least active units in the neighborhood */
    lowest=99999999.9;
    highest=(-1.0);
    for(i=lowi; i<=highi; i++)

```



```

        (squaresumnowhere-sumnowhere*sumnowhere/nsamples));
printf(" G %.2f, %.2f\n", sumgotnone/nsamples,
       sqrt(1.0/(nsamples-1.0)*
           (squaresumgotnone-sumgotnone*sumgotnone/nsamples)));
}

float distance2d(x1,y1,x2,y2)
float x1,y1,x2,y2;
{
/* unit response is equal to the euclidian distance of input and weight */
return(sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)));
}

```

E.5.2 Data files

The simulation parameters and data are specified in the simufile. Below is the simufile for the first test, storing and retrieving a particular vector (figures 8.1 and 8.2). The storefile consists of the vector 0.4 0.4, and the cuefile of the vector 0.7 0.5.

```

storefile          ;;: vectors to be store
cuefile           ;;: cue vectors
9 0.1 1           ;;: nets, lowact, displaying
1 1 1            ;;: ngrid, ntraces, nsamples
25 1.4 10.0 2.0 -0.5 ;;: tsettle, minact, maxact, gammaexc, gammainh

```

In the basin simulation (figure 8.3), the same simufile was used, but the code was conditionalized to draw the cues from the grid. The storefile contained the following vectors:

```

0.6 0.7
0.8 0.5
0.2 0.2

```

In the performance descriptor simulation of figure 8.4, slightly different parameters were used. The storefile and the testfile are redundant, since the vectors to be stored are random and the cues were obtained from a grid.

```

storefile          ;;: vectors to be store
testfile          ;;: cue vectors
20 0.1 1          ;;: nets, lowact, displaying
20 10 10         ;;: ngrid, ntraces, nsamples
25 1.4 15.0 0.5 -0.075 ;;: tsettle, minact, maxact, gammaexc, gammainh

```

```

    ngotnone++;
    ngotnoneperc = 100.0*ngotnone/traces;
    sumgotnone += ngotnoneperc;
    squaresumgotnone += ngotnoneperc*ngotnoneperc;
}

count(traces)
int traces;
{
    int i,besttrace;
    float dist,distance2d(), best;
    /* nothing retrieved */
    if(besti==(-1))
    {
        there=(-1);
        nnowhere++;
    }
    else
    {
        there=(-2);
        /* find the retrieved vector */
        for(i=0; i<traces; i++)
            if(tracecenter[i].i==besti && tracecenter[i].j==bestj)
            {
                /* it was one of the stored ones */
                there=i;
                tracecenter[i].gotanyyet=1;
            }

        if(there==(-2)) /* it was not any of the stored ones */
            nbogus++;
        else /* it was one of the stored ones */
        {
            /* find the traces that was closest to the cue */
            besttrace=(-1);
            best=999999999.9;
            for(i=0; i<traces; i++)
            {
                dist=distance2d(x,y,units[tracecenter[i].i][tracecenter[i].j][1],
                                units[tracecenter[i].i][tracecenter[i].j][2]);
                if(dist < best)
                {
                    besttrace=i; best=dist;
                }
            }
            if(besttrace==there) /* the closest one was retrieved */
                nnearest++;
            else /* the closest was not retrieved */
                nanother++;
        }
    }
}

print_trace_stats(traces)
int traces;
{
    printf("%d traces:\n",traces);
    printf("N %.2f %.2f", sumnearest/nsamples,
           sqrt(1.0/(nsamples-1.0)*
                (squaresumnearest-sumnearest*sumnearest/nsamples)));
    printf("A %.2f %.2f", sumanother/nsamples,
           sqrt(1.0/(nsamples-1.0)*
                (squaresumanother-sumanother*sumanother/nsamples)));
    printf("B %.2f %.2f", sumbogus/nsamples,
           sqrt(1.0/(nsamples-1.0)*
                (squaresumbogus-sumbogus*sumbogus/nsamples)));
    printf("N %.2f, %.2f", sumnowhere/nsamples,
           sqrt(1.0/(nsamples-1.0)*

```

```

float sigmoid(activity)
/* transform the activity to a sigmoid response between 0 and maximum
   equation 8.2 */
float activity;
{
    return(1.0/(1.0+exp(maxact*(minact-*activity))));
}

modify_weights()
/* create the trace on the lateral weights emanating from units
   within the neighborhood; equation 8.3 */
{
    register int i,j, ii, jj;
    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
            if(units[i][j][0] > lowact)
                for(ii=0; ii<nets; ii++)
                    for(jj=0; jj<nets; jj++)
                        if((units[ii][jj][0] > units[i][j][0]+epsilon) ||
                           (ii == i && jj == j))
                            latweights[i][j][ii][jj] = gammaexc*units[i][j][0];
                        else
                            latweights[i][j][ii][jj] = gammainh*units[i][j][0];;
}

init_system(argc,argv)
/* set up the input weight vectors on a regular grid */
int argc; char *argv[];
{
    register int i,j;

    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
            {
                units[i][j][1] = (i+1.0)/(nets+1.0);
                units[i][j][2] = (j+1.0)/(nets+1.0);
            }
    init_latweights();
}

collect_sample_stats(traces)
int traces;
{
    int i, ngotnone;
    float npoints,
    nnearestperc, nanotherperc, nbogusperc, nnowhereperc, ngotnoneperc;

    npoints = (float) ngrid*ngrid;          /* points in the grid */

    nnearestperc = 100.0*nnearest/npoints;
    nanotherperc = 100.0*nanother/npoints;
    nbogusperc = 100.0*nbogus/npoints;
    nnowhereperc = 100.0*nnowhere/npoints;

    sumnearest += nnearestperc;
    sumanother += nanotherperc;
    sumbogus += nbogusperc;
    sumnowhere += nnowhereperc;

    squaresumnearest += nnearestperc*nnearestperc;
    squaresumanother += nanotherperc*nanotherperc;
    squaresumbogus += nbogusperc*nbogusperc;
    squaresumnowhere += nnowhereperc*nnowhereperc;

    ngotnone=0;                               /* traces that were not retrieved */
    for(i=0; i<traces; i++)
        if(tracecenter[i].gotanyet == 0)

```

```

        units[i][j][0] = 0.0;
    }
    /* let the network settle */
    bestvalue=1.0;
    for(ts=1; (ts<=tsettle && bestvalue>=aliveact) || ts<=tsettle/2; ts++)
    /* if activity drops < aliveact after tsettle/2, the map is oscillating:
       no use continuing until tsettle */
        compute_lat_responses();

    if(bestvalue<aliveact)          /* nothing retrieved */
        besti=bestj=(-1);
}

compute_lat_responses()
{
    register int i,j;
    float best=0.0;
    float unit_resp();
    /* copy the current response to be the previous response */
    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
        {
            units[i][j][4] = units[i][j][0];
        }
    /* compute the new responses with theta=0.5 (retrieval) */
    bestvalue=compute_unit_responses(0.5);
}

float compute_unit_responses(theta)
/* compute the responses of all units in the map and find the max */
float theta;
{
    register int i,j;
    float unit_resp();
    float best=0.0;
    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
        {
            units[i][j][0] = unit_resp(i,j,theta);
            /* check if this unit's response is maximum so far encountered */
            if (units[i][j][0] > best)
            {
                besti=i; bestj=j; best=units[i][j][0];
            }
        }
    return(best);
}

float unit_resp(ui,uj,theta)
/* compute the response of a single unit; equation 8.1 */
int ui,uj;
float theta;
{
    register int i,j, lowi, highi, lowj, highj;
    float extresp=latresp=0.0, distance2d(),sigmoid();
    /* external input activation */
    extresp = sqrt(2.0) - distance2d(x,y,units[ui][uj][1],units[ui][uj][2]);
    /* lateral interaction */
    if (theta>0.0)
        for(i=0; i<nets; i++)
            for(j=0; j<nets; j++)
            {
                latresp = latresp + latweights[i][j][ui][uj] * units[i][j][4];
            }
    return(sigmoid((1.0-theta)*extresp+theta*latresp));
}

```

```

        for(t=0; t<traces; t++)
        {
            generate_input(t);          /* get the vector to be stored */
            compute_unit_responses(0);
            tracecenter[t].i=besti;     /* record the center of the response */
            tracecenter[t].j=bestj;
            if(newvector(besti,bestj,t)) /* if it has not occurred before */
                modify_weights();       /* create the trace */
            else t--;                   /* try again */
        }
    #if test==0
        for(t=0; t<ntests; t++)
    #else
        for(t=0; t<ngrid*ngrid; t++)
    #endif
        {
            generate_tests(t);          /* get the cue vector */
            settle(t);                 /* relax the response */
            count(traces);             /* when test==3, collect statistics */
        }
        collect_sample_stats(traces);
    }
    print_trace_stats(traces);
}
exit(0);
}

generate_input(t)
int t;
{
    float f01rnd();                   /* see D.1.2 for listing of f01rnd() */
    #if test=2
        /* generating random vectors to be stored */
        x=f01rnd();
        y=f01rnd();
    #else
        /* specific vectors, specified in the simulation file */
        x=xs[t];
        y=ys[t];
    #endif
}

generate_tests(t)
int t;
{
    /* input uniformly distributed in 2-D unit square */
    float f01rnd();
    /* generating random cues */
    /* x=f01rnd();
       y=f01rnd(); */
    #if test=0
        /* specific vectors, specified in the simulation file */
        x=testxs[t];
        y=testys[t];
    #else
        /* cues from a regular grid */
        x = (0.5+t/ngrid)/ngrid;
        y = (0.5+t%ngrid)/ngrid;
    #endif
}

settle(t)
register int t;
{
    register int i,j,ts;
    /* for each input, start with a blank network */
    for(i=0; i<nets; i++)
        for(j=0; j<nets; j++)
            {

```


E.5.1 Code

Three different tests were performed with this code: (1) storing and retrieving a particular vector (figures 8.1 and 8.2), (2) measuring the basins of three traces (figure 8.3), and (3) measuring performance descriptors with increasing number of traces (figure 8.3). All these tests were performed by the conditionalizing the code below. As usual, graphics, initializations and file-I/O are not listed.

```
#include <stdio.h>
#include <math.h>

#define test=2          /* 0=store and cue vectors specified in a file
                        1=store v. in a file, cue v. in a grid (basins)
                        2=random store v., cue v. in a grid (descriptors) */

#define epsilon 0.0001
#define aliveact 0.5   /* oscillation / retrieval */

#define maxnets 21    /* maximum network size is n*n units */
#define maxinputs 100 /* maximum number of store vectors */
#define maxtests 100  /* maximum number of cue vectors */

/* network size, image unit indices, settling, grid and simulation params */
int nets, besti, bestj, ntests, there, tsettle, ngrid, ntraces, nsamples;

/* sigmoid and lateral weight parameters */
float minact, maxact, lowact, gammaexc, gammainh,
float bestvalue; /* activity of the maximally responding unit */

/* statistics variables for test==3 */
float sumnearest, sumanother, sumbogus, sumnowhere, sumgotnone,
squaresumnearest, squaresumanother, squaresumbogus, squaresumnowhere,
squaresumgotnone;
int nnearest, nanother, nbogus, nnowhere;
struct {int i,j,gotanyyet} tracecenter[maxinputs];

/* units and weights */
float units[maxnets][maxnets][5],
latweights[maxnets][maxnets][maxnets][maxnets];

/* store and cue vectors (2-dimensional) */
float x, y, xs[maxinputs], ys[maxinputs],
testxs[maxtests], testys[maxtests];

/* simulation spec. file */
char simufile[100];

main(argc,argv)
int argc; char *argv[];
{
    register int t,traces,sample;
    int i;
    sprintf(simufile,"%s", argv[1]);
    init_system(argc,argv); /* set up the map */
    read_params(); /* not listed: read simulation specs */

    /* when test==3, try increasing number of traces */
    for(traces=1; traces<=ntraces; traces++)
    {
        init_trace_stats();
        /* when test==3, collect data from several trials */
        for(sample=1; sample<=nsamples; sample++)
        {
            init_latweights(); /*not listed;initially all inhibitory*/
            init_sample_stats(traces);/*not included:clear stats for the sample*/
            /* store each trace */

```

```

lreps          ;; lexical representations
sreps          ;; semantic representations
inp            ;; inputfile
0 0 1 150      ;; displaying,testing,seed, tend
9 0 50 150 4 1 0.1 0.05 ;; lnets,lt0,lt1,lt2,lnct0,lnct1,lalphat0,lalphat1
7 0 50 150 4 1 0.1 0.05 ;; snets,st0,st1,st2,snct0,snct1,salphat0,salphat1
  0 50 150 0 0 0.1 0.05 ;;          at0,at1,at2,foo,foo,aalphat0,aalphat1
150 999999999 ;; snapshots (epoch):-1=initial, 9x9=last
...
0.445206 0.665961 0.149145 0.444724 0.482224 0.703811 0.665403
...
0.737084 0.575349 0.998192 0.779288 0.629970 0.737808 0.189245 0.412983
0.002025 0.835223 0.536956 0.747543
...
0.506770 ... 0.252776 ... 0.000014 ... 0.000012 ... 0.000024 ...
0.245148 ... 0.245148
...

```

The training data (file inp) consists of lexical symbol - semantic concept pairs. There is one pair for each synonymous and homonymous sense of each word in the input file, e.g. CHICKEN -- prey and CHICKEN -- food. Again, the actual file contains indices instead of words.

```

ATE      ate
BROKE   broke
HIT     hit
...
CHICKEN prey
CHICKEN food
...
BALL    gear
BAT     gear
...

```

The file lreps contains the lexical representations and the file sreps contains the semantic representations. Below is the top of each file, containing the representations for ATE and ate. The representations are graphically displayed in figure 9.2.

```

lvoc          ;; lexical word labels (used for display only)
7             ;; lnrep
0.4814815 0.33333334 0.7037037 0.0 0.0 0.0 0.0
...

svoc          ;; semantic word labels (used for display only)
12           ;; snrep
0.000000 0.913352 0.710347 0.297255 0.720355 0.988037 0.724369 0.523655
0.968736 0.939058 0.006160 0.021069
...

```

E.5 Ideal trace feature maps

The properties of trace feature maps were demonstrated with an ideal implementation (section 8.1) which is slightly different from the implementation of the trace feature maps in DISCERN (appendix D.1.6. The code is listed below in section E.5.1, and the simulation files are listed in section E.5.2.

```

(* step (- num minval)))
(defun spit-out (allwords)
  (dolist (word allwords)
    (dolist (letter word)
      (format t "~&F" letter))))

```

The input data consists of pairs of symbols and concepts. The mapping between them is many-to-many in this data. The training set is generated by the following LISP program.

```

(setq lwords
  '(((_ . -1)
    (ATE . 0)
    (BROKE . 1)
    (HIT . 2)
    ;;etc ...
    (LION . 29)
    ))

(setq swords
  '(((_ . -1)
    (ate . 0)
    (broke . 1)
    (hit . 2)
    ;;etc ...
    (prey . 16)
    ))

(setq pairs
  '((ATE . ate) (BROKE . broke) (HIT . hit)
    (MOVED . moved) (BALL . hitter1) (BAT . livebat)
    (BAT . hitter1) (BOY . human) (PAPERWT . hitter2)
    (CHEESE . food) (CHICKEN . food) (CHICKEN . prey)
    (CURTAIN . furnitu) (DESK . furnitu) (DOLL . doll)
    (FORK . utensil) (GIRL . human) (HATCHET . hitter1)
    (HAMMER . hitter1) (MAN . human) (WOMAN . human)
    (PLATE . fragile) (ROCK . hitter2) (PASTA . food)
    (SPOON . utensil) (CARROT . food) (VASE . vase)
    (WINDOW . fragile) (DOG . dog) (WOLF . predatr)
    (SHEEP . prey) (LION . predatr)
    ))

;;; get a pair and output the indices
(defun gener ()
  (dolist (pair pairs)
    (format t "~&S ~S"
      (cdr (assoc (car pair) lwords))
      (cdr (assoc (cdr pair) swords)))))

```

E.4.2 Data files

Below is a sample of the simulation file, showing the weight representation for the lexical symbol CHICKEN (unit (0,7) in figure 9.3) and the semantic concept prey (unit (0,1) in figure 9.4).

The last 7 numbers are weights on the associative connections from CHICKEN to the second row from the bottom of the semantic map (figure 9.4). The strongest connection, 0.506770, is to unit labeled prey, and there are weaker connections to the next unit on the row and to the units next to food on the other side of the map.

E.4 Sentence data for the lexicon

The same code was used to develop the lexicon for the sentence processing data (section 9.3) as for DISCERN (appendix C.3), only the data files were different.

E.4.1 Generating training data

The lexical representations are generated according to the darkness values of the letters in the Macintosh Geneva font, using the LISP program below.

```
;;; vocabulary
(setq voc '(
ATE          BROKE          HIT          MOVED          BALL
BAT          BOY           PAPERWT      CHEESE         CHICKEN
CURTAIN      DESK           DOLL         FORK           GIRL
HATCHET      HAMMER         MAN          WOMAN          PLATE
ROCK         PASTA          SPOON        CARROT         VASE
WINDOW       DOG            WOLF         SHEEP          LION
))

(setq letters '(
(ATE . (A T E _ _ _)) (BROKE . (B R O K E _ _))
(HIT . (H I T _ _ _)) (MOVED . (M O V E D _ _))
(BALL . (B A L L _ _)) (BAT . (B A T _ _))
(BOY . (B O Y _ _)) (PAPERWT . (P A P E R W T))
(CHEESE . (C H E E S E _)) (CHICKEN . (C H I C K E N))
(CURTAIN . (C U R T A I N)) (DESK . (D E S K _ _))
(DOLL . (D O L L _ _)) (FORK . (F O R K _ _))
(GIRL . (G I R L _ _)) (HATCHET . (H A T C H E T))
(HAMMER . (H A M M E R _)) (MAN . (M A N _ _))
(WOMAN . (W O M A N _ _)) (PLATE . (P L A T E _ _))
(ROCK . (R O C K _ _)) (PASTA . (P A S T A _ _))
(SPOON . (S P O O N _ _)) (CARROT . (C A R R O T _ _))
(VASE . (V A S E _ _)) (WINDOW . (W I N D O W _))
(DOG . (D O G _ _)) (WOLF . (W O L F _ _))
(SHEEP . (S H E E P _ _)) (LION . (L I O N _ _))
))

;;; these are counts of dark pixels of the letter
(setq values
'(( . 5) (A . 18) (B . 27) (C . 17) (D . 24) (E . 24) (F . 17)
(G . 25) (H . 23) (I . 9) (J . 13) (K . 18) (L . 13) (M . 32)
(N . 23) (O . 22) (P . 21) (Q . 23) (R . 24) (S . 19) (T . 14)
(U . 19) (V . 15) (W . 27) (X . 17) (Y . 12) (Z . 19)
))

;;; scale the letter values between 0 and 1
(setq minval 5)
(setq maxval 32)
(setq step (/ 1 (- maxval minval)))

(defun gener ()
  (spit-out (mapcar 'convert voc)))

(defun convert (word)
  (mapcar 'getvalue (explode word)))

(defun explode (word)
  (cdr (assoc word letters)))

(defun getvalue (letter)
  (scale (cdr (assoc letter values))))

(defun scale (num)
```

E.3.1 Mapping of the FGREP representations

Below are the data and simulation specifications for the feature mapping of FGREP representations (figure 4.9). The start times for middle and bottom levels below are later than the end of the simulation, i.e. only the top level is organized.

```
basicreps          ;;; repfile
basicinp           ;;; inputfile
testing           ;;; testfile (not used)
0 0 1 15000 0.001 0.001      ;;; displaying,testing,seed,minvar0,minvar1
10 1000 15000 4 1 0.1 0.05   ;;; t: nets, t1,t2,nct0,nct1,alphan0,alphan1
0 6000 10 2500 1 0 0.02 0.01 ;;; m: nets,t0,t1,t2,nct0,nct1,alphan0,alphan1
0 12000 500 2500 6 1 0.2 0.1 ;;; b: nets,t0,t1,t2,nct0,nct1,alphan0,alphan1
15000 999999999          ;;; snapshots (epoch):-1=initial, 9x9=last
15000
0.544813 0.125000 0.701004 0.105072 0.533508 0.551966 0.380970 0.529911
0.274356 0.625062 0.354034 0.549179 ...
...
```

The file basicreps contains the 12-dimensional representation vectors (representation of ball is the first one of the 24 vectors in the file):

```
24 12
0.257738 0.889293 0.263792 0.860573 0.533735 0.497301 0.124718 0.398592
0.899659 0.005595 0.411466 0.099813
...
```

The input file basicinp now consists of word labels and representation indices, with number of slots = 1:

```
1          ;;; nslot
ball
0
bat
1
boy
2
...
```

E.3.2 Single-level mapping of script-based stories

The single-level feature mapping of figure 6.2 was developed and tested with the same training data as the hierarchical feature maps. Again, only the top level is included in the simulation by setting the start times for middle and bottom levels after the end of the simulation:

```
reps            ;;; repfile
inp            ;;; inputfile
testing       ;;; testfile (not used)
0 0 12345 300 0.001 0.001      ;;; displaying,testing,seed,minvar0,minvar1
24 25 550 14 1 0.2 0.1        ;;; t: nets, t1,t2,nct0,nct1,alphan0,alphan1
0 6000 10 2500 1 0 0.02 0.01 ;;; m: nets,t0,t1,t2,nct0,nct1,alphan0,alphan1
0 12000 500 2500 6 1 0.2 0.1 ;;; b: nets,t0,t1,t2,nct0,nct1,alphan0,alphan1
300 999999999          ;;; snapshots (epoch):-1=initial, 9x9=last
300
0.157958 0.644636 0.000270 0.083585 0.933675 0.149510 0.180672 0.148669
0.085472 0.073499 0.924721 0.020148 ...
...
```

```

}

float mindist(c1, c2)
/* find the smallest distance between items in two clusters */
int c1[],c2[];
{
    register int i,j;
    /* see D.2.1 for listing of distance() */
    float dist, min = 999999999.9, distance();
    for(i=1; i<=c1[0]; i++)
        for(j=1; j<=c2[0]; j++){
            dist=distance(vectors[c1[i]].comp, vectors[c2[j]].comp, dim);
            if(dist<min) min=dist;
        }
    return(min);
}

printclusters(count)
int count;
{
    register int i,j;
    printf("\nClusters at iteration %d:\n", count);
    for(i=0; i<nvectors; i++)
        if(clusters[i][0]>0){
            printf("%d: ", i);
            for(j=1; j<=clusters[i][0]; j++)
                printf("%s ", vectors[ clusters[i][j]].label);
            printf("\n");
        }
}

```

E.2.2 Data files

The input file for merge clustering consists of labelfilename, dimension, and the vectors. The first representation, for ball, is shown below.

```

voc                ;;:labelfile
12
0.257738 0.889293 0.263792 0.860573 0.533735 0.497301 0.124718 0.398592
0.899659 0.005595 0.411466 0.099813
...

```

The labelfile contains the actual words:

ball	bat	boy	paperwt	cheese
chicken	doll	fork	girl	hatchet
hammer	man	woman	plate	rock
pasta	spoon	carrot	vase	window
dog	wolf	sheep	lion	

E.3 Single-level feature maps

The single level feature maps for the FGREP representations (figure 4.9) and for the script data (figure 6.2 and section 7.3.2) were developed with the hierarchical feature map code, by disabling the hierarchy and self-organizing only the top level.

E.2 Merge clustering

The hierarchical merge clustering algorithm used in forming optimal clusters of FGREP representations (figure 4.6) is presented below.

E.2.1 Code

```
#include <stdio.h>
#include <math.h>

#define maxvectors 600          /* maximum number of input vectors */
#define maxlabell 30          /* maximum length of labels */
#define maxdim 50             /* maximum dimension of input vectors */

int besti, bestj, dim, nvectors,
clusters[maxvectors][maxvectors+1]; /* initially each word is a cluster,
/* clusters[i][0] counts the number
of words in the cluster */

/* files */
char vectorfile[100], labelfile[100];
FILE *fp,*fp2;

struct inputvectors {
    char label[maxlabell+1];
    float comp[maxdim];
} vectors[maxvectors];

main(argc,argv)
int argc; char *argv[];
{
    float mindist(), best, dist;
    register int i,j, count=0;
    sprintf(vectorfile, "%s", argv[1]);
    fp=fopen(vectorfile,"r");
    read_params(fp); /* not included;read labels and vectors,init clusters*/
    fclose(fp);
    printclusters(0);
    while (count<nvectors-1)
    {
        /* find the minimum linkage */
        best=99999999.9;
        for(i=0; i<nvectors; i++)
            if(clusters[i][0]>0)
                for(j=i+1; j<nvectors; j++)
                    if(clusters[j][0]>0){
                        dist=mindist(clusters[i], clusters[j]);
                        if(dist<best){
                            besti=i; bestj=j; best=dist;
                        }
                    }
        printf("Merging clusters %d and %d: %f\n", besti, bestj, best);
        /* move items from the second cluster to the first */
        for(i=clusters[bestj][0]+1; i<=clusters[besti][0]+clusters[bestj][0]; i++)
            clusters[besti][i]=clusters[bestj][i-clusters[besti][0]];
        /* update number of items in this cluster */
        clusters[besti][0]=clusters[besti][0]+clusters[bestj][0];
        /* the second cluster is now empty */
        clusters[bestj][0]=0;
        printclusters(++count);
    }
    fclose(fp);
    exit(0);
}
```

A representation for each of these instances is added to the simufle. For example, when making 36 clones of each word, and spacing the IDs evenly on the unit square (0.2 increments), the first three instances of `ate` are:

```
0.000000 0.000000 0.512749 0.070246 0.027442 0.575880 0.778024 0.997699
0.579190 0.036904 0.756611 0.574002   ;;; ate1
0.000000 0.200000 0.512749 0.070246 0.027442 0.575880 0.778024 0.997699
0.579190 0.036904 0.756611 0.574002   ;;; ate2
0.000000 0.400000 0.512749 0.070246 0.027442 0.575880 0.778024 0.997699
0.579190 0.036904 0.756611 0.574002   ;;; ate3
...
```

The test data is generated separately for each different number of clones (see section E.1.3).

E.1.6 Sequential input

The experiments with sequential-input FGREP in section 4.6 were performed with the sentence parser module of DISCERN (appendix sections E.1.1 and D.1.3). Because the input consists of actual sequences of words, three entries were added to the basic case-role assignment vocabulary of sections E.1.4 and E.1.5: “.” (period), `the` and `with`. The data generator program (section E.1.3) was modified to include these words:

```
print_sentence(genno)
int genno;
{
  if(obj!=(-1))          /* if there is object, output 'the' before it */
    theobj=the;
  if(with!=(-1))        /* same for the with-clause */
  {
    thewith=the;
    withwith=with0;
  }
  printf("1 %d %d %d %d %d %d %d %d %d %d %d %d (%d.) %s %s %s %s\n",
        the,subj,verb,theobj,obj,withwith,thewith,with,period,
        agent,verb,patient,instr,mod, genno,
        words[subj], words[verb], words[obj], words[with]);
}

all_blanks()
/* default all case roles to empty */
{
  /* first the, subject, verb and period are always present */
  theobj=(-1); obj=(-1); withwith=(-1); thewith=(-1); with=(-1);
  agent=(-1); patient=(-1); instr=(-1); mod=(-1);
}

```

The sentence data now has the form:

```
0 9 5 1          ;;; nslot, nword, ncase, nsent
1 31 6 2 31 6 32 31 4 30 6 2 6 -1 4 (13.) human hit human gear
1 31 6 2 31 6 32 31 4 30 6 2 6 4 -1 (14.) human hit human gear
```

where 31 = `the`, 32 = `with` and 30 = `period`.

When cloned synonyms are used (section 4.5), only one of the synonymous words appears in the training data. For example, instead of man, woman, boy, girl there are only indices for boy (which actually stands for human now) in the training set. The final representation for that word serves as the word prototype. The above training data is replaced by:

```

4 5                               ;;; ninpas, noutas
6 2 6 4 6 2 6 -1 4 (13.) human hit human hitter1
6 2 6 4 6 2 6 4 -1 (14.) human hit human hitter1
...

```

E.1.5 Testing data

Basic FGREP performance (without cloning synonyms) is tested using the same simulation files and vocabulary files as for training. The input files have the same format as above, but contain different sentence sets (e.g. familiar and unfamiliar sentences, tables 4.3 and 4.4, or all sentences, table 4.5).

For testing performance with cloned synonyms, the prototype representation such as human is cloned to cover the whole 30 word vocabulary and the cloned representations are put in the simulation file. For example, there are following representations for man, woman, boy, girl in the simulation file during testing:

```

...
0.000000 0.000000 0.016987 0.792115 0.111307 0.950957 0.015306 0.934702
0.029477 0.103723 0.032938 0.019578 ;;; boy
...
0.000000 1.000000 0.016987 0.792115 0.111307 0.950957 0.015306 0.934702
0.029477 0.103723 0.032938 0.019578 ;;; girl
...
1.000000 0.000000 0.016987 0.792115 0.111307 0.950957 0.015306 0.934702
0.029477 0.103723 0.032938 0.019578 ;;; man
1.000000 1.000000 0.016987 0.792115 0.111307 0.950957 0.015306 0.934702
0.029477 0.103723 0.032938 0.019578 ;;; woman
...

```

For tests with extended vocabulary, every word in the data is developed as a word prototype. For testing, a new vocabulary is created, which contains entries for each of the new instances:

ate	broke	hit	moved	hitter1
bat	human	hitter2	food	chicken
curtain	desk	doll	utensil	.
.	.	.	.	fragile
.	.	.	.	vase
.	dog	predator	sheep	.
.	the	with	ate1	ate2
ate3	ate4	ate5	ate6	ate7
ate8	ate9	ate10	ate11	ate12
ate13	ate14	ate15	ate16	ate17
ate18	ate19	ate20	ate21	ate22
ate23	ate24	ate25	ate26	ate27
ate28	ate29	ate30	ate31	ate32
ate33	ate34	ate35	ate36	broke1
broke2	broke3	broke4	broke5	broke6
...				

```
4      ;;; npredator
458 459 460 461
```

E.1.4 Training data

As with training multiple FGREP modules (appendix C.1), the simulation file initially contains only 9 lines specifying the data files, simulation parameters and snapshot iterations to be saved. Each snapshot is saved at the end of the simulation file. A snapshot contains the current word representations and the current weights for the FGREP network. Shown below is the simufile with only the final configuration stored, showing the first two word representations (**ate** and **broke**), and the beginning of the weights.

```

voc                ;;; wordfile
traininp           ;;; inputfile
12 25              ;;; nwordrep, nhidrep
0 0 1 3           ;;; displaying, testing, seed, nphase
200 500 600       ;;; (+ ) lasttimes(epochs) for each phase
0.1 0.05 0.025    ;;; eta for each phase
600 999999999     ;;; snapshots time(epoch):-1=initial, 9x9=last
600
0.475054 0.999724 0.184593 0.122073 0.719814 0.615625 0.561436 0.689919
0.606559 0.999458 0.998683 0.596351
0.310299 0.134998 0.903760 0.511304 0.121276 0.228168 0.365528 0.383097
0.222569 0.082273 0.334186 0.233612
...
-0.455774 0.726841 -0.967763 0.056370 -0.438670 0.489411 0.847474
-0.538549 -0.683143 0.133675 ...
...

```

The training set of sentence data is read from the file `traininp` (generated by the gener program in section E.1.3). Below are a few sample entries from this file. Each entry consists of word indices, which refer to the word representations in the simulation file above, and they also index the file `voc` which contains the labels for the words.

```

4 5                ;;; ninpas, noutas
6 2 14 4 6 2 14 -1 4 (13.) boy hit girl ball
6 2 14 4 6 2 14 4 -1 (14.) boy hit girl ball
...

```

The first four indices indicate the syntactic constituents of the sentence (spelled out in words at the end of the line), and their representations are concatenated to form the input pattern for the network. The next five indices specify the target case-role assignment. The number in parenthesis shows the generator number (referring to table 4.1). The two entries above have the same input sentence, **the boy hit the girl with the ball** but in the first sentence **ball** is the modifier, in the second one it is the instrument. A snapshot of the network processing the latter entry is shown in figure 4.4.

There are 30 words in the training vocabulary (file `voc`):

ate	broke	hit	moved	ball
bat	boy	paperwt	cheese	chicken
curtain	desk	doll	fork	girl
hatchet	hammer	man	woman	plate
rock	pasta	spoon	carrot	vase
window	dog	wolf	sheep	lion

```

1      ;;; nmoved
3
26     ;;; nthing
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
18     ;;; nobject
4 5 7 8 9 10 11 12 13 15 16 19 20 21 22 23 24 25
7      ;;; nhitter
5 4 15 16 24 7 20
6      ;;; nbreaker
5 4 15 16 7 20
7      ;;; npossession
5 4 15 16 24 26 12
6      ;;; nanimal
5 9 28 28 27 29
2      ;;; nprey
9 28
4      ;;; nfood
9 8 21 23
3      ;;; nfragile
19 25 24
4      ;;; nhuman
6 14 17 18
2      ;;; nutensil
13 22
2      ;;; npredator
27 29

```

When the vocabulary is extended by cloning several synonymous instances of each word, the test data is generated separately for each different number of clones. For example, the following data file specifies the word categories when four clones are created for each word:

```

exp25voc
4      ;;; nate
33 34 35 36
4      ;;; nbroke
58 59 60 61
4      ;;; nhit
83 84 85 86
4      ;;; nmoved
108 109 110 111
60     ;;; nthing
133 134 135 136 158 159 160 161 183 184 185 186 208 209 210 211
233 234 235 236 258 259 260 261 283 284 285 286 308 309 310 311
333 334 335 336 358 359 360 361 383 384 385 386 408 409 410 411
433 434 435 436 458 459 460 461 483 484 485 486 44
      ;;; nobject
133 134 135 136 158 159 160 161 208 209 210 211
233 234 235 236 258 259 260 261 283 284 285 286 308 309 310 311
333 334 335 336 358 359 360 361 383 384 385 386 408 409 410 411
16     ;;; nhitter
133 134 135 136 158 159 160 161 208 209 210 211 408 409 410 411
12     ;;; nbreaker
133 134 135 136 158 159 160 161 208 209 210 211
20     ;;; npossession
133 134 135 136 158 159 160 161 333 334 335 336 408 409 410 411
433 434 435 436
20     ;;; nanimal
158 159 160 161 258 259 260 261 433 434 435 436 458 459 460 461
483 484 485 486
8      ;;; nprey
258 259 260 261 483 484 485 486
8      ;;; nfood
258 259 260 261 233 234 235 236
8      ;;; nfragile
383 384 385 386 408 409 410 411
4      ;;; nhuman
183 184 185 186
4      ;;; nutensil
358 359 360 361

```

```

/* generator 1. */
all_blanks();
for(h=0; h<nate; h++)
{
    verb=ate[h];
    for(i=0; i<nhuman; i++)
    {
        subj=human[i];
        agent=subj;
        print_sentence(1);
    }
}

/* etc... */

/* generator 14. */
all_blanks();
for(h=0; h<nhit; h++)
{
    verb=hit[h];
    for(i=0; i<nhuman; i++)
    {
        subj=human[i];
        agent=subj;
        for(j=0; j<nthing; j++)
        {
            obj=thing[j];
            patient=obj;
            for(k=0; k<nhitter; k++)
            {
                with=hitter[k];
                instr=with;
                print_sentence(14);
            }
        }
    }
}

/* etc... */
}

print_sentence(genno)
int genno;
{
    printf("%d %d %d %d %d %d %d %d %d (%d.) %s %s %s %s\n",
        subj,verb,obj,with,agent,verb,patient,instr,mod, genno,
        words[subj], words[verb], words[obj], words[with]);
}

all_blanks()
/* default all case roles to empty */
{
    /* the verb and the subject are always specified */
    obj=(-1); with=(-1);
    agent=(-1); patient=(-1); instr=(-1); mod=(-1);
}

```

The word categories for training (table 4.2) are specified in the following file:

```

voc      ;;; labelfile
1        ;;; nate
0
1        ;;; nbroke
1
1        ;;; nhit
2

```

E.1.3 Generating data

The input data is generated by the program below. File I/O has been omitted from the listing, and only the generators 1. and 14. are included (see table 4.1).

```
#include <stdio.h>
#include <math.h>

#define maxwords 750          /* maximum size of vocabulary */
#define maxwordl 30          /* max length of the word labels */
#define maxclone 36          /* max number of synonyms for each word */

#define maxate (1*maxclone)  /* max number of items in the categories */
#define maxbroke (1*maxclone)
#define maxhit (1*maxclone)
#define maxmoved (1*maxclone)
#define maxthing (15*maxclone)
#define maxobject (11*maxclone)
#define maxhitter (4*maxclone)
#define maxbreaker (3*maxclone)
#define maxpossession (5*maxclone)
#define maxanimal (5*maxclone)
#define maxprey (2*maxclone)
#define maxfood (2*maxclone)
#define maxfragile (2*maxclone)
#define maxhuman (1*maxclone)
#define maxutensil (1*maxclone)
#define maxpredator (1*maxclone)

/* word labels */
char blank[maxwordl]=" "; /* blank word */
char words[maxwords][maxwordl];

/* files */
char datafile[100], wordfile[100];
FILE *fp;

/* indices for words in each category */
int ate[maxate], broke[maxbroke], hit[maxhit], moved[maxmoved],
thing[maxthing], object[maxobject], hitter[maxhitter],
breaker[maxbreaker], possession[maxpossession], animal[maxanimal],
prey[maxprey], food[maxfood], fragile[maxfragile], human[maxhuman],
utensil[maxutensil], predator[maxpredator];

/* actual number of words in each category */
int nate, nbroke, nhit, nmoved, nthing, nobject, nhitter, nbreaker,
npossession, nanimal, nprey, nfood, nfragile, nhuman, nutensil, npredator;

/* these variables are assigned differently for each sentence */
int verb, subj, obj, with, tverb, agent, patient, instr, mod;

main(argc,argv)
int argc; char *argv[];
{
    sprintf(datafile, "%s", argv[1]);
    read_data(); /* not included; read the category data */
    read_words(); /* read the labels */
    printf("4 5 ;;; ninpas, noutas\n");
    generate_sentences();
    exit(0);
}

generate_sentences()
{
    register int h, i, j, k;
```

```

    if (itemized_display)
    {
        for(i=0; i<5+ninpas*8+maxdam*(maxerr*6+6); i++)
            printf("-");
        printf("\n");
    }
}

print_footer()
/* averages for the whole epoch */
{
    register int i,j;
    if (itemized_display)
    {
        for(i=0; i<5+ninpas*8+maxdam*(maxerr*6+6); i++)
            printf("-");
        printf("\n");
    }
    printf("AVERAGE %29s", " ");
    for(i=0; i<maxdam; i++)
    {
        printf("%6.3f", deltasum[i]/(noutas*noutrep*nsents));
        for(j=0; j<maxerr; j++)
            printf("%6.1f", 100.0*totnerr[i][j]/(noutas*noutrep*nsents));
    }
    printf("\n");
}

print_words()
/* list of words and hits for the whole epoch */
{
    register int i,j;
    int correcttok;
    printf(" \nAll words:   out of %d, correct", noutas*nsents);
    for(i=0; i<maxdam; i++)
        printf(" %d (%3.1f%%) ",
            correctwords[i], 100.0*correctwords[i]/(noutas*nsents));
    printf(" \nSynon. words: out of %d, correct", nalltok );
    for(i=0; i<maxdam; i++)
    {
        correcttok = correctwords[i] -
            testwordscorrect[i][-1] - testwordscorrect[i][0] -
            testwordscorrect[i][1] - testwordscorrect[i][2] -
            testwordscorrect[i][3] - testwordscorrect[i][5] -
            testwordscorrect[i][9] - testwordscorrect[i][10] -
            testwordscorrect[i][11] - testwordscorrect[i][12] -
            testwordscorrect[i][24] - testwordscorrect[i][26] -
            testwordscorrect[i][28] - testwordscorrect[i][30] -
            testwordscorrect[i][31] - testwordscorrect[i][32];
        printf(" %d (%3.1f%%) ", correcttok,
            100.0*correcttok/nalltok);
    }
    printf("\n");
    for(i=(-1); i<nwords; i++)
        if (testwordsal1[i]>0)
        {
            printf("%10s: %2d ", words[testwords[i]].chars, testwordsal1[i]);
            for (j=0; j<maxdam; j++)
                printf(" %2d", testwordscorrect[j][i]);
            printf("\n");
        }
}

```

```

int d;
{
    register int i,j,k;
    int nerr[maxerr];
    for(i=0; i<maxerr; i++) nerr[i]=0.0;
    for(i=0; i<noutas; i++)
        for(j=0; j<noutrep; j++)
            for(k=0; k<maxerr; k++)
                if (fabs(outrep[i][j]-tchrep[i][j]) < err[k]) nerr[k]++;

    for(i=0; i<maxerr; i++)
    {
        if (itemized_display) printf("%.1f", 100.0*nerr[i]/(noutas*noutrep));
        totnerr[d][i] += nerr[i];
    }
}

cumulate_error(d)
/* cumulate data for average error per unit */
int d;
{
    register int i,j;
    float locdsum=0.0;
    for(i=0; i<noutas; i++)
        for(j=0; j<noutrep; j++)
            locdsum += fabs(tchrep[i][j]-outrep[i][j]);
    if (itemized_display) printf("%.3f", locdsum/(noutas*noutrep));
    deltasum[d] += locdsum;
}

cumulate_nearest(senti,d)
/* cumulate data for counting the number of words nearest to the correct
   lexicon entry */
int senti,d;
{
    register int i,j;
    int correctnum=0;
    for(i=0; i<noutas; i++)
        /* determine_nearest is similar to find_nearest, see D.1.2 */
        if(determine_nearest(outrep[i], noutrep) == tchnums[senti][i])
        {
            correctnum++;
            testwordscorrect[d][ tchnums[senti][i] ]++;
        }
    correctwords[d] += correctnum;
}

/***** print out results *****/

print_header()
{
    register int i,j;
    printf("\n\nInput file: %s  Repr.size: %d  Hidden layer size: %d\n",
        inputfile, nwordrep, nhidrep);
    printf("Phase: %-2d Epoch: %-2d Eta: %-.4.2f Damage%%: ",
        phase+1, epoch, eta);
    for(i=0; i<maxdam; i++)
    {
        printf("%.1f", 100.0*((int) (damage[i]*nwordrep))/nwordrep);
        for(j=0; j<6*maxerr; j++) printf(" ");
    }
    printf("\nGarno Input %7s Within%%: %8s", " ", " ");
    for(i=0; i<maxdam; i++)
    {
        printf("%6s", " ");
        for(j=0; j<maxerr; j++) printf("%.1f", 100.0*err[j]);
    }
    printf("\n");
}

```

```

        randfun(&words[7].rep[j], 0.0, 1.0);
        randfun(&words[8].rep[j], 0.0, 1.0);
        randfun(&words[13].rep[j], 0.0, 1.0);
        randfun(&words[14].rep[j], 0.0, 1.0);
        randfun(&words[15].rep[j], 0.0, 1.0);
        randfun(&words[16].rep[j], 0.0, 1.0);
        randfun(&words[17].rep[j], 0.0, 1.0);
        randfun(&words[18].rep[j], 0.0, 1.0);
        randfun(&words[19].rep[j], 0.0, 1.0);
        randfun(&words[20].rep[j], 0.0, 1.0);
        randfun(&words[21].rep[j], 0.0, 1.0);
        randfun(&words[22].rep[j], 0.0, 1.0);
        randfun(&words[23].rep[j], 0.0, 1.0);
        randfun(&words[25].rep[j], 0.0, 1.0);
        randfun(&words[27].rep[j], 0.0, 1.0);
        randfun(&words[29].rep[j], 0.0, 1.0);
    }
}

/* use this when randomizing all words */
/*randomize_tokens()
{
    register int i,j;

    for (i=4; i<nwords; i++)
        for(j=0; j<2; j++)
            randfun(&words[i].rep[j], 0.0, 1.0);
}*/
#endif

/***** fwdprop *****/

forward_prop(nokinprep)
/* bias units were not used in this simulation, compare to section C.1.1 */
int nokinprep;
{
    register int i,j,k,p;
    float sigmoid();

    for(k=0; k<nhidrep; k++)
    {
        hidrep[k]=0.0;
        for(i=0; i<ninpas; i++)
        {
            for(j=0; j<nokinprep; j++) /* leave out units from the end */
                hidrep[k] += inprep[i][j]*wih[i][j][k];
            for(j=nokinprep; j<ninprep; j++)
                hidrep[k] += 0.5*wih[i][j][k];
        }
        hidrep[k]=sigmoid(hidrep[k]);
    }
    for(i=0; i<noutas; i++)
        for(j=0; j<noutrep; j++)
        {
            outrep[i][j]=0;
            for(k=0; k<nhidrep; k++)
                outrep[i][j] += hidrep[k]*who[i][j][k];
            outrep[i][j] = sigmoid(outrep[i][j]);
        }
}

/***** collect data *****/

cumulate_within(d)
/* cumulate data for counting units within error range */

```



```

/* go through one epoch */
{
    register int i,j;
    get_current_params();
    init_snapshot();
    print_header();
    for(i=0; i<nsects; i++)
        sentence(i);
    print_footer();
    print_words();
}

sentence(senti)
/* process one sentence */
register int senti;
{
    register int i,j;
    int *indataptr;

#if toks
    /* when testing with random IDs, randomize tokens here */
    /*randomize_tokens();*/          /* randomize each sentence separately */
#endif
    /* get the current input and target patterns from the lexicon */
    for (i=0; i<ninpas; i++)
        for(j=0; j<ninprep; j++)
            inprep[i][j]=words[inpnums[senti][i]].rep[j];
    for(i=0; i<noutas; i++)
        for(j=0; j<noutrep; j++)
            tchrep[i][j]=words[tchnums[senti][i]].rep[j];
    propagate_and_print(senti);
}

propagate_and_print(senti)
register int senti;
{
    register int d,i,j,k;
    int nokinprep; /* upper bound for units included in the rep */
    float sigmoid();

    if (itemized_display)
    {
        printf(" %2d. ",geners[senti]);
        for(i=0; i<ninpas; i++) printf("%-8s", words[inpnums[senti][i]].chars);
    }
    /* test separately for each level of damage */
    for(d=0; d<maxdam; d++)
    {
        nokinprep=ninprep-((int) (damage[d]*ninprep));
        forward_prop(nokinprep);
        cumulate_error(d);
        cumulate_nearest(senti,d);
        cumulate_within(d);
    }
    if (itemized_display) printf("\n");
}

#if toks
/* use this when randomizing the synonymous words */
randomize_tokens()
{
    register int i,j;

    for(j=0; j<2; j++)
    {
        randfun(&words[4].rep[j], 0.0, 1.0);
        randfun(&words[6].rep[j], 0.0, 1.0);
    }
}

```

```

blank, testwords[maxwords],
blankall, testwordsall[maxwords],
blankcorrect, testwordscorrect[maxdam][maxwords+1],nalltok;

/* error ranges and damages tested */
float err[maxerr]={0.15},
      damage[maxdam]={0.0},
/* damage[maxdam]={0.0, 0.1, 0.2, 0.3, 0.4, 0.45, 0.55,
0.6, 0.7, 0.8, 0.9, 0.95, 1.0},*/
      totnerr[maxdam][maxerr], deltasum[maxdam];

/* representation indices */
int inpnums[maxsents][maxinpas], tchnums[maxsents][maxoutas];

/* units and weights */
float inprep[maxinpas][maxrep], outrep[maxoutas][maxrep],
      tchrep[maxoutas][maxrep], hidrep[maxrep],
      wih[maxinpas][maxrep][maxrep], who[maxoutas][maxrep][maxrep];

/* filenames */
char simufile[100], wordfile[100], inputfile[100];
FILE *fp;

/* lexicon */
char blankchars[maxwordl]; /* simply blank (which is a word also) */
float blankrep[maxrep]; /* blankrep=words[-1].rep */
struct lexicon {
    char chars[maxwordl];
    float rep[maxrep];
} words[maxwords];

/***** main control *****/
main(argc,argv)
int argc; char *argv[];
{
    sprintf(simufile, "%s", argv[1]);
    fp=fopen(simufile,"r");
    read_params(fp); /* (not listed) read the simulation specs */
    fclose(fp);
    sprintf(inputfile, "%s", argv[2]);
    read_inputs(); /* (not listed) read the test data */
    init_test(); /* (not listed) count targets */
    iterate_snapshots();
    exit(0);
}

iterate_snapshots()
/* go through the saved snapshots, collecting performance statistics */
{
    int readfun();

    fp=fopen(simufile,"r");
    read_params(fp);

    while (fscanf(fp,"%d",&epoch)!=EOF) /* read the epoch */
    {
        iterate_weights(readfun); /* read the current word reps and weights */
        srand(seed); /* we want to test the same ID:s for all snapshots */
        iterate_inputs(); /* collect statistics for one epoch */
    }
    fclose(fp);
}

iterate_inputs()

```

```

/* modify representations; equations 4.11 and 4.12 */
for(i=0; i<ninpas; i++)
  if (inpnms[senti][i]>-1)
    for(j=0; j<ninprep; j++)
      words[inpnms[senti][i]].rep[j] = inprep[i][j] =
        clip(inprep[i][j] +eta*inpsumsig[i][j]);
}

/***** math stuff *****/

float sigmoid(activity)
float activity;
{
  /* transform the activity to a sigmoid response between 0 and 1
  equation 4.1 */
  return(1.0/(1.0+exp(-activity)));
}

float clip(activity)
/* cut activity within 0 and 1; equation 4.12 */
float activity;
{
  if (activity<0.0) return(0.0);
  else if (activity > 1.0) return(1.0);
  else return(activity);
}

```

E.1.2 Testing code

The performance tests for basic FGREP were run with the program below. File I/O and initializations have been omitted from the listing.

```

#include <stdio.h>
#include <math.h>

#define snapshotend 999999999
#define toks 0 /* tokens / no */

/* max table dimensions */
#define maxrep 75 /* maximum size of the representations */
#define maxsents 2000 /* maximum number of sentences */
#define maxwordl 30 /* maximum length of input words (chars) */
#define maxinpas 9 /* maximum number of input assemblies */
#define maxoutas 9 /* maximum number of output assemblies */
#define maxsnaps 50 /* maximum number of snapshots */
#define maxphase 5 /* maximum number of phases */
#define maxwords 750 /* maximum number of lexicon entries */

/* some parameters for testing */
#define blanklength 0.15 /* a vector shorter than this is considered blank */
#define itemized_display 0 /* 1=statistics output for each sentence */
#define maxerr 1 /* # of different error ranges tested */
#define maxdam 1 /* # of degrees of damage tested */
/*#define maxdam 13*/

/* simulation parameters */
int fildes, nphase, ninpas, noutas, ninprep, noutrep, nwordrep, nhidrep,
  nwords, phase, epoch, geners[maxsents], nsents, seed;
long phaseends[maxphase], snapshots[maxsnaps];
float colors[256][3], etas[maxphase], eta;

/* vars for collecting statistics */
int correctwords[maxdam],

```

```

float sigmoid();

/* from input to hidden layer; equation 4.1 */
for(k=0; k<nhidrep; k++)
{
    hidrep[k]=0.0;
    for(i=0; i<ninpas; i++)
        for(j=0; j<ninprep; j++)
            hidrep[k] += inprep[i][j]*wih[i][j][k];
    hidrep[k]=sigmoid(hidrep[k]);
}

/* from hidden to output layer; 4.1 */
for(i=0; i<noutas; i++)
    for(j=0; j<noutrep; j++)
    {
        outrep[i][j]=0.0;
        for(k=0; k<nhidrep; k++)
            outrep[i][j] += hidrep[k]*who[i][j][k];
        outrep[i][j] = sigmoid(outrep[i][j]);
    }
}

backward_prop(senti)
register int senti;
{
    register int i,j,k;
    float clip(), sig, fact, hidsumsig[maxrep], inpsumsig[maxinpas][maxrep];

    /* clear the error signals */
    for(k=0; k<nhidrep; k++)
        hidsumsig[k]=0.0;
    for(i=0; i<ninpas; i++)
        for(j=0; j<ninprep; j++)
            inpsumsig[i][j]=0.0;

    for(i=0; i<noutas; i++)
        for(j=0; j<noutrep; j++)
        {
            /* error signal at the output layer; equation 4.3 */
            sig = (tchrep[i][j]-outrep[i][j])*
                outrep[i][j]*(1.0-outrep[i][j]);
            fact=eta*sig;
            for(k=0; k<nhidrep; k++)
            {
                /* cumulate error signal at the hidden layer; equation 4.5 */
                hidsumsig[k] += sig*who[i][j][k];
                /* modify output weights; equation 4.6 */
                who[i][j][k] += fact*hidrep[k];
            }
        }

    for(k=0; k<nhidrep; k++)
    {
        /* error signal at the hidden layer; equation 4.5 */
        sig = hidsumsig[k]*hidrep[k]*(1.0-hidrep[k]);
        fact = eta*sig;
        for(i=0; i<ninpas; i++)
            for(j=0; j<ninprep; j++)
            {
                /* cumulate error signal at the input layer; equation 4.10 */
                inpsumsig[i][j] += sig*wih[i][j][k];
                /* modify input weights; equation 4.6 */
                wih[i][j][k] += fact*inprep[i][j];
            }
    }
}

```

```

/***** main control *****/
/* the simulation control functions main(), iterate_snapshots()
and training() are similar to ones listed in C.1.1 and have been omitted */
iterate_inputs()
{
    int i;
    for(i=0; i<nSENTS; i++)
        sentence(shuffletable[i]);
}

sentence(senti)
/* present one input sentence */
register int senti;
{
    register int i,j;
    int *inpdataptr;
#ifdef TOKS
    randomize_tokens(); /* set up the IDs randomly for the story */
#endif
    /* get the current input and teaching patterns from the lexicon */
    for (i=0; i<nINPAS; i++)
        for(j=0; j<nINPREP; j++)
            inprep[i][j]=words[inpnums[senti][i]].rep[j];
    for(i=0; i<nOUTAS; i++)
        for(j=0; j<nOUTREP; j++)
            tchrep[i][j]=words[tchnums[senti][i]].rep[j];
    forward_prop();
    if (!testing) backward_prop(senti); /* when testing, don't change weights */
}

#ifdef TOKS
/* use this when cloning the synonymous words */
randomize_tokens()
{
    register int i,j;

    for(j=0; j<2; j++)
    {
        randfun(&words[6].rep[j], 0.0, 1.0);
        randfun(&words[13].rep[j], 0.0, 1.0);
        randfun(&words[27].rep[j], 0.0, 1.0);
        randfun(&words[19].rep[j], 0.0, 1.0);
        randfun(&words[4].rep[j], 0.0, 1.0);
        randfun(&words[7].rep[j], 0.0, 1.0);
        randfun(&words[8].rep[j], 0.0, 1.0);
    }
}

/* use this when cloning all words */
/*randomize_tokens()
{
    register int i,j;

    for (i=4; i<nWORDS; i++)
        for(j=0; j<2; j++)
            randfun(&words[i].rep[j], 0.0, 1.0);
}*/
#endif

/***** propagation *****/

forward_prop()
/* bias units were not used in this simulation, compare to section C.1.1 */
{
    register int i,j,k,p;

```

APPENDIX E

Analysis and comparison code and data

E.1 Case-role assignment with FGREP

E.1.1 Training code

The basic non-recurrent FGREP network for the sentence case-role assignment data (chapter 4) is trained with the program below. Graphics, initializations and file I/O have been omitted from the listing.

```
#include <stdio.h>
#include <math.h>

#define toks 0                /* tokens / no */

/* max table dimensions */
#define maxrep 50             /* maximum size of the representations */
#define maxsents 2000        /* maximum number of sentences */
#define maxwordl 20          /* maximum length of word labels (chars) */
#define maxinpas 9           /* maximum number of input assemblies */
#define maxoutas 9           /* maximum number of output assemblies */
#define maxsnaps 50          /* maximum number of snapshots */
#define maxphase 5           /* maximum number of phases */
#define maxwords 100         /* maximum number of lexicon entries */
#define snapshotend 99999999 /* this is the last in the list of snapshots */

/* simulation parameters */
int fildes, displaying, continuing, testing, nextsnapshot,
    nphase, startepoch, epoch, phaseends[maxphase],
    seed, cmapsize, shuffletable[maxsents];
float colors[256][3], etas[maxphase], eta,
long phaseends[maxphase], snapshots[maxsnaps];

/* actual table dimensions used */
int ninpas, noutas, ninprep, noutrep, nhidrep, nwords, nwordrep, nsents;

/* representation indices */
int inpnums[maxsents][maxinpas], tchnums[maxsents][maxoutas];

/* units and weights */
float inprep[maxinpas][maxrep], outrep[maxoutas][maxrep],
    tchrep[maxoutas][maxrep], hidrep[maxrep],
    wih[maxinpas][maxrep][maxrep], who[maxoutas][maxrep][maxrep];

/* file names */
char simufile[100], wordfile[100], inputfile[100];
FILE *fp;

/* lexicon */
float blankrep[maxrep];      /* blankrep=words[-1].rep */
struct lexicon {
    char chars[maxwordl];
    float rep[maxrep];
} words[maxwords];
```

who ate _ _ fish denny's WHO-QUESTION
 1 john did _ _ _ _ WHO-ANSWER
 john did _ _ _ _ WHO-ANSWER

\$restaurant \$fancy mary steak leone's good big
 1 who ate steak _ _ _ _ ?
 who ate _ _ steak leone's WHO-QUESTION
 1 mary did _ _ _ _ WHO-ANSWER
 mary did _ _ _ _ WHO-ANSWER

\$restaurant \$coffee john fish denny's bad small
 1 what did john eat _ _ _ _ ?
 john ate _ _ what denny's WHAT-QUESTION
 1 john ate a bad fish _ _ _ _
 john ate _ bad fish _ WHAT-ANSWER

\$shopping \$electronics john tv radioshack _ _
 1 who bought tv _ _ _ _ ?
 who bought _ _ tv radioshack WHO-QUESTION
 1 john did _ _ _ _ WHO-ANSWER
 john did _ _ _ _ WHO-ANSWER

\$shopping \$electronics mary tv circuitcty _ _
 1 who bought tv at circuitcty _ _ _ _ ?
 who bought _ _ tv circuitcty WHO-QUESTION
 1 mary did _ _ _ _ WHO-ANSWER
 mary did _ _ _ _ WHO-ANSWER

\$restaurant \$fancy mary steak leone's good big
 1 how did mary like lobster at leone's ?
 mary thought _ what steak leone's HOW-QUESTION
 1 mary thought steak was good at leone's
 mary thought _ good steak leone's HOW-ANSWER

\$travel \$plane john dfw jfk big _
 1 what did john take to jfk _ _ ?
 john took _ _ what jfk WHAT-QUESTION
 1 john took a plane _ _ _ _
 john took _ _ plane _ WHAT-ANSWER

0 mary waited at the gate for boarding .
 mary waited _ _ boarding gate WAITING
 0 mary got-on the plane _ _ _ .
 mary got-on _ _ _ plane GETTING-ON
 0 the plane took-off from dfw _ _ _ .
 plane took-off _ _ _ dfw TAKING-OFF
 1 the plane arrived at sfo _ _ _ .
 plane arrived _ _ _ sfo ARRIVING
 0 mary got-off the plane _ _ _ .
 mary got-off _ _ _ plane GETTING-OFF

\$restaurant \$fancy mary steak leone's good big
 1 what did mary eat at leone's _ ?
 mary ate _ _ what leone's WHAT-QUESTION
 1 mary ate a good steak _ _ .
 mary ate _ _ good steak _ WHAT-ANSWER

\$restaurant \$fancy mary steak leone's good big
 1 how did mary like steak at leone's ?
 mary thought _ _ what steak leone's HOW-QUESTION
 1 mary thought steak was good at leone's .
 mary thought _ _ good steak leone's HOW-ANSWER

\$travel \$plane mary dfw sfo big _
 1 where did mary take a plane to ?
 mary took _ _ plane where WHERE-QUESTION
 1 mary took a plane to sfo _ .
 mary took _ _ plane sfo WHERE-ANSWER

\$travel \$plane mary dfw sfo big _
 1 did mary travel big distance to sfo ?
 mary traveled _ _ big distance sfo DID-QUESTION
 1 mary traveled a big distance _ _ .
 mary traveled _ _ big distance _ DID-ANSWER

\$restaurant \$coffee john fish denny's bad small
 1 where did john eat fish _ _ ?
 john ate _ _ fish where WHERE-QUESTION
 1 john ate fish at denny's _ _ .
 john ate _ _ fish denny's WHERE-ANSWER

\$shopping \$electronics mary tv circuitcty _ _
 1 where did mary buy tv _ _ ?
 mary bought _ _ tv where WHERE-QUESTION
 1 mary bought tv at circuitcty _ _ .
 mary bought _ _ tv circuitcty WHERE-ANSWER

\$shopping \$electronics john tv radioshack _ _
 1 where did john buy tv _ _ ?
 john bought _ _ tv where WHERE-QUESTION
 1 john bought tv at radioshack _ _ .
 john bought _ _ tv radioshack WHERE-ANSWER

At this point in the example run, the search threshold parameter was changed from 1.0 to 3.0 to allow DISCERN to look at several maps for the answer for the following ambiguous questions. In the data, the sequence of input words is ambiguous, but the story representation and the case-role representation for the question and answer sentences specify complete representations (they could be anything, since they are only used for comparison in collecting performance data in the performance phase).

1 1 1 0 ;;; chained, withlex, withhfm, context_mode
 1 1 ;;; babbling, print_mistakes
 0 7 ;;; nstories, nquest
 \$restaurant \$coffee john fish denny's bad small
 1 who ate _ _ _ _ ?


```

0 john left a small tip - - - .
  john left waiter small tip - TIPPING
0 john paid the cashier - - - .
  john paid cashier - - - PAYING
0 john left denny's - - - .
  john left - - - denny's LEAVING

```

Next DISCERN reads four more stories and answers seven questions about the five stories now in its memory. Now context_mode=0), indicating that memory retrieval is performed for each question.

```

1 1 1 0      ;;; chained, withlex, withfm, context_mode
1 1          ;;; babbling, print_mistakes
4 7          ;;; nstories, nquest
$restaurant $fancy mary steak leone's good big
1 mary went to leone's - - - .
  mary went - - - leone's ENTERING
0 the waiter seated mary - - - .
  waiter seated - - - mary SEATING
1 mary asked the waiter for steak - - - .
  mary asked waiter - steak ORDERING
0 mary ate a good steak - - - .
  mary ate - good steak EATING
0 mary paid the waiter - - - .
  mary paid waiter - - - PAYING
1 mary left a big tip - - - .
  mary left waiter big tip TIPPING
0 mary left leone's - - - .
  mary left - - - leone's LEAVING

$shopping $electronics mary tv circuitcty - -
1 mary went to circuitcty - - - .
  mary went - - - circuitcty ENTERING
1 mary looked for good tv - - - .
  mary looked - good tv LOOKING
0 mary asked the salesperson questions about tv .
  mary asked salesperson tv questions DECIDING
1 mary took the best tv - - - .
  mary took - best tv TAKING
0 mary paid the cashier - - - .
  mary paid cashier - - - PAYING
0 mary left circuitcty - - - .
  mary left - - - circuitcty LEAVING
-1 - - - - - FILLER

$shopping $electronics john tv radioshack - -
1 john went to radioshack - - - .
  john went - - - radioshack ENTERING
1 john looked for good tv - - - .
  john looked - good tv LOOKING
0 john asked the salesperson questions about tv .
  john asked salesperson tv questions DECIDING
1 john took the best tv - - - .
  john took - best tv TAKING
0 john paid the cashier - - - .
  john paid cashier - - - PAYING
0 john left radioshack - - - .
  john left - - - radioshack LEAVING
-1 - - - - - FILLER

$travel $plane mary dfw sfo big -
1 mary went to dfw - - - .
  mary went - - - dfw ORIGIN
1 mary checked-in for a flight to sfo .
  mary checked-in - sfo flight CHECKING-IN

```

```

    who bought _ _ cd-player radioshack WHO-QUESTION
1 mary did _ _ _ _ WHO-ANSWER
    mary did _ _ _ _ WHO-ANSWER
1 what did mary buy at radioshack _ ?
    mary bought _ _ what radioshack WHAT-QUESTION
1 mary bought cd-player _ _ _ _
    mary bought _ _ cd-player _ WHAT-ANSWER
1 where did mary buy cd-player _ _ ?
    mary bought _ _ cd-player where WHERE-QUESTION
1 mary bought cd-player at radioshack _ _
    mary bought _ _ cd-player radioshack WHERE-ANSWER
-1 _ _ _ _ _
    _ _ _ _ _ FILLER
-1 _ _ _ _ _
    _ _ _ _ _ FILLER

```

```

$travel $plane john lax sfo big _
1 john went to lax _ _ _ _
    john went _ _ _ _ lax ORIGIN
1 john checked-in for a flight to sfo .
    john checked-in _ sfo flight _ CHECKING-IN
0 john waited at the gate for boarding .
    john waited _ _ boarding gate WAITING
0 john got-on the plane _ _ _ _
    john got-on _ _ _ _ plane GETTING-ON
0 the plane took-off from lax _ _ _ _
    plane took-off _ _ _ _ lax TAKING-OFF
1 the plane arrived at sfo _ _ _ _
    plane arrived _ _ _ _ sfo ARRIVING
0 john got-off the plane _ _ _ _
    john got-off _ _ _ _ plane GETTING-OFF
1 who traveled to sfo _ _ _ _ ?
    who traveled _ _ _ _ sfo WHO-QUESTION
1 john did _ _ _ _ _
    john did _ _ _ _ _ WHO-ANSWER
1 what did john take to sfo _ ?
    john took _ _ what sfo WHAT-QUESTION
1 john took a plane _ _ _ _
    john took _ _ _ _ plane WHAT-ANSWER
1 where did john take a plane to ?
    john took _ _ _ _ plane where WHERE-QUESTION
1 john took a plane to sfo _ _ _ _
    john took _ _ _ _ plane sfo WHERE-ANSWER
1 did john travel big distance to sfo ?
    john traveled _ big distance sfo DID-QUESTION
1 john traveled a big distance _ _ _ _
    john traveled _ big distance _ DID-ANSWER

```

D.3.4 Example run

The example run presented in section 10.2 consisted of processing three input files in a single session. The first input file contains a three sentence story which is paraphrased immediately (`context_mode=1`), without questions (`nquest=0`).

```

1 1 1 1      ;;; chained, withlex, withhfm, context_mode
1 1          ;;; babbling, print_mistakes
1 0          ;;; nstories, nquest
$restaurant $coffee john fish denny's bad small
1 john went to denny's _ _ _ _
    john went _ _ _ _ denny's ENTERING
0 john seated john _ _ _ _
    john seated _ _ _ _ john SEATING
1 john asked the waiter for fish _ _
    john asked waiter _ fish ORDERING
1 john ate a bad fish _ _ _ _
    john ate _ bad fish _ EATING

```

```

john took _ _ plane where WHERE-QUESTION
1 john took a plane to sfo _ .
john took _ _ plane sfo WHERE-ANSWER
1 did john travel big distance to sfo ?
john traveled _ big distance sfo DID-QUESTION
1 john traveled a big distance _ .
john traveled _ big distance _ DID-ANSWER

```

D.3.3 Incomplete stories

The “incomplete story” test set consists of the same stories, but now included is 0 for some of the sentences, indicating that these sentences are not part of the input:

```

1 1 1 1      ;;; chained, withlex, withhfm, context_mode
1 1          ;;; babbling, print_mistakes
1 0          ;;; nstories, nquest
$restaurant $fancy john lobster mamaison good big
1 john went to mamaison _ _ .
john went _ _ _ mamaison ENTERING
0 the waiter seated john _ _ .
waiter seated _ _ john SEATING
1 john asked the waiter for lobster _ .
john asked waiter _ lobster ORDERING
0 john ate a good lobster _ .
john ate _ good lobster EATING
0 john paid the waiter _ .
john paid waiter _ _ _ PAYING
1 john left a big tip _ _ .
john left waiter big tip _ TIPPING
0 john left mamaison _ _ .
john left _ _ _ mamaison LEAVING
1 who ate lobster at mamaison _ _ ?
who ate _ _ lobster mamaison WHO-QUESTION
1 john did _ _ _ _ .
john did _ _ _ _ WHO-ANSWER
1 what did john eat at mamaison _ ?
john ate _ _ what mamaison WHAT-QUESTION
1 john ate a good lobster _ _ .
john ate _ good lobster WHAT-ANSWER
1 where did john eat lobster _ _ ?
john ate _ _ lobster where WHERE-QUESTION
1 john ate lobster at mamaison _ _ .
john ate _ _ lobster mamaison WHERE-ANSWER
1 how did john like lobster at mamaison ?
john thought _ what lobster mamaison HOW-QUESTION
1 john thought lobster was good at mamaison .
john thought _ good lobster mamaison HOW-ANSWER

$shopping $electronics mary cd-player radioshack _ _
1 mary went to radioshack _ _ .
mary went _ _ _ radioshack ENTERING
1 mary looked for good cd-player _ _ .
mary looked _ good cd-player LOOKING
0 mary asked the salesperson questions about cd-player .
mary asked salesperson cd-player questions DECIDING
1 mary took the best cd-player _ _ .
mary took _ best cd-player TAKING
0 mary paid the cashier _ _ .
mary paid cashier _ _ _ PAYING
0 mary left radioshack _ _ .
mary left _ _ _ radioshack LEAVING
-1 _ _ _ _ _ FILLER
1 who bought cd-player at radioshack _ _ ?

```

john did _ _ _ WHO-ANSWER
 1 what did john eat at mamaison _ ?
 john ate _ _ what mamaison WHAT-QUESTION
 1 john ate a good lobster _ _
 john ate _ _ good lobster WHAT-ANSWER
 1 where did john eat lobster _ _ ?
 john ate _ _ lobster where WHERE-QUESTION
 1 john ate lobster at mamaison _ _
 john ate _ _ lobster mamaison WHERE-ANSWER
 1 how did john like lobster at mamaison ?
 john thought _ what lobster mamaison HOW-QUESTION
 1 john thought lobster was good at mamaison .
 john thought _ good lobster mamaison HOW-ANSWER

\$shopping \$electronics mary cd-player radioshack _ _
 1 mary went to radioshack _ _
 mary went _ _ radioshack ENTERING
 1 mary looked for good cd-player _ _
 mary looked _ _ good cd-player LOOKING
 1 mary asked the salesperson questions about cd-player .
 mary asked salesperson cd-player questions DECIDING
 1 mary took the best cd-player _ _
 mary took _ _ best cd-player TAKING
 1 mary paid the cashier _ _
 mary paid cashier _ _ PAYING
 1 mary left radioshack _ _
 mary left _ _ radioshack LEAVING
 -1 _ _ _ _ FILLER
 1 who bought cd-player at radioshack _ _ ?
 who bought _ _ cd-player radioshack WHO-QUESTION
 1 mary did _ _ _ _
 mary did _ _ _ _ WHO-ANSWER
 1 what did mary buy at radioshack _ ?
 mary bought _ _ what radioshack WHAT-QUESTION
 1 mary bought cd-player _ _ _ _
 mary bought _ _ cd-player WHAT-ANSWER
 1 where did mary buy cd-player _ _ ?
 mary bought _ _ cd-player where WHERE-QUESTION
 1 mary bought cd-player at radioshack _ _
 mary bought _ _ cd-player radioshack WHERE-ANSWER
 -1 _ _ _ _ FILLER
 -1 _ _ _ _ FILLER
 -1 _ _ _ _ FILLER

\$travel \$plane john lax sfo big _
 1 john went to lax _ _ _ _
 john went _ _ _ _ lax ORIGIN
 1 john checked-in for a flight to sfo .
 john checked-in _ sfo flight CHECKING-IN
 1 john waited at the gate for boarding .
 john waited _ _ boarding gate WAITING
 1 john got-on the plane _ _ _ _
 john got-on _ _ _ _ plane GETTING-ON
 1 the plane took-off from lax _ _ _ _
 plane took-off _ _ _ _ lax TAKING-OFF
 1 the plane arrived at sfo _ _ _ _
 plane arrived _ _ _ _ sfo ARRIVING
 1 john got-off the plane _ _ _ _
 john got-off _ _ _ _ plane GETTING-OFF
 1 who traveled to sfo _ _ _ _ ?
 who traveled _ _ _ _ sfo WHO-QUESTION
 1 john did _ _ _ _
 john did _ _ _ _ WHO-ANSWER
 1 what did john take to sfo _ ?
 john took _ _ what sfo WHAT-QUESTION
 1 john took a plane _ _ _ _
 john took _ _ plane WHAT-ANSWER
 1 where did john take a plane to ?

```

0.84375 0.53125 0.28125 0.4375 0.75 0.78125 0.0 0.0 0.0 0.0
0.53125 0.53125 0.59375 0.71875 0.28125 0.75 0.78125 0.0 0.0 0.0
...

```

The srepfile is exactly the same as the semantic representation file used to train the hierarchical feature maps (section C.2.2). It includes FGREP representations for the script and the track names, developed by the FGREP modules along with the other semantic representations (see section C.1). Compared to the FGREP training representations, the word prototypes have been replaced by instances, e.g. PERSON has been replaced by John (with IDs 0.75, 0.25) and Mary (with 0.25, 0.75). \$restaurant appears on top of the file below, and the two other vectors stand for John and Mary.

```

svoc ;; svocfile
12 ;; snrep
0.065524 0.533577 0.006305 0.024564 0.038476 0.513618 0.000749 0.059773
0.612141 0.998206 0.170227 0.000885
...
0.750000 0.250000 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
0.250000 0.750000 0.277490 0.965935 0.214716 0.987744 0.013766 0.987437
0.485509 0.150869 0.991646 0.995925
...

```

D.3.2 Complete stories

The “complete story” test set consists of the data produced by the generators of section D.2. The included flag (the number preceding each sentence) is always 1, indicating that all sentences are included in the input story. Below are a few samples from the “complete story” test set. These are instances of the same training stories that were shown in section C.1 (see that section for an explanation of the format). As in the training, the actual datafiles for DISCERN consist of word indices instead of words. The indices refer to lexical and semantic representations, which are kept in separate files, and to the actual lexical and semantic words, also in separate files. Below, the actual words are listed instead of indices for legibility.

```

1 1 1 1      ;;; chained, withlex, withhfm, context_mode
1 1          ;;; babbling, print_mistakes
1 0          ;;; nstories, nquest
$restaurant $fancy john lobster mamaison good big
1 john went to mamaison _ _ _ _
  john went _ _ _ mamaison ENTERING
1 the waiter seated john _ _ _ _
  waiter seated _ _ john _ SEATING
1 john asked the waiter for lobster _ _
  john asked waiter _ lobster _ ORDERING
1 john ate a good lobster _ _ _
  john ate _ good lobster _ EATING
1 john paid the waiter _ _ _ _
  john paid waiter _ _ _ PAYING
1 john left a big tip _ _ _ _
  john left waiter big tip _ TIPPING
1 john left mamaison _ _ _ _
  john left _ _ _ mamaison LEAVING
1 who ate lobster at mamaison _ _ _ ?
  who ate _ _ lobster mamaison WHO-QUESTION
1 john did _ _ _ _ _

```

foo	foo	foo	foo	foo
foo	foo	foo	foo	foo
foo	foo	foo	foo	foo
foo	foo	foo	foo	foo
foo	foo	foo	foo	foo
foo	foo	foo	foo	foo
foo	foo	foo	foo	foo
STAFF	DRIVER	WAITER	CASHIER	CONDUCTOR
WAITED	ATE	WENT	SEATED	ASKED
PUNCHED	TRAVELED	TASTED	PAID	LEFT
TOOK	CHECKED-IN	LOOKED	TRIED	COMPARED
OFF	BOUGHT	TOOK-OFF	GOT-ON	ARRIVED
SEVERAL	PRICES	TIP	NONE	LINE
SMALL	BEST	GOOD	BAD	BIG
FLIGHT	BOARDING	QUESTIONS	CART	GATE
BUS	DISTANCE	TICKET	PLANE	TRAIN
ABOUT	FOR	.	A	THE
ON	IN	FROM	TO	AT
WHERE	HOW	?	WHO	WHAT
THOUGHT	WAS	DID	EAT	LIKE
JOHN	MARY	BUY	TAKE	TRAVEL
STEAK	DENNY'S	MAMAISON	LEONE'S	LOBSTER
MCDONALD'S	BURGERKING	NORMS	SPAGHETTI	FISH
BULLOCK'S	SHOES	HAMBURGER	FRIES	BROADWAY
PLAYER	TV	JEANS	RADIOSHACK	CIRCUITCTY
VEGETABLES	LAX	RALPH'S	SAFEWAY	SOFTDRINKS
CENTRALSTA	DOWNTOWNST	DFW	JFK	SFO
BUS-STOP	DOWNTOWN	NEWYORK	BOSTON	BUS-TERMIN
		BEACH		

Semantic word labels:

\$restaurant	\$fancy	\$coffee	\$fast	\$shopping
\$clothing	\$electronics	\$grocery	\$travel	\$plane
\$train	\$bus	waiter	cashier	conductor
staff	driver	went	seated	asked
waited	ate	tasted	paid	left
punched	traveled	looked	tried	compared
took	checked-in	took-off	got-on	arrived
got-off	bought	tip	none	line
several	prices	good	bad	big
small	best	questions	cart	gate
flight	boarding	ticket	plane	train
bus	distance	.	a	the
about	for	from	to	at
on	in	?	who	what
where	how	did	eat	like
thought	was	buy	take	travel
John	Mary	MaHaison	Leone's	lobster
steak	Denny's	Norms	spaghetti	fish
McDonald's	BurgerKing	hamburger	fries	Broadway
Bullock's	shoes	jeans	RadioShack	CircuitCty
CD-player	tv	Ralph's	Safeway	softdrinks
vegetables	lax	dfw	jfk	sfo
CentralSta	DowntownSt	NewYork	Boston	bus-termin
bus-stop	downtown	beach		

Lrepfile and srepfile contain the representations for the lexical and semantic words. The lrepfile is basically the same as the one used to train the lexicon, except the foos are padded with 1.0, making them dissimilar to any real word in the data. The first actual representations are WAITER and CASHIER, shown below.

```
lvoc ;; pvocfile
10   ;; pnpref
1.0 1.0 ...
...
```

```

(dolist (item '(softdrinks vegetables))
  (dolist (store '(ralph\s safeway))
    (print-out (grocery-list person item store))))))

;;; generate travel stories
(defun generplane ()
  (dolist (person '(john mary))
    (dolist (origin '(lax dfw))
      (dolist (dest '(jfk sfo))
        (print-out (plane-list person origin dest))))))

(defun genertrain (distance)
  (dolist (person '(john mary))
    (dolist (origin '(centralsta downtownst))
      (dolist (dest '(boston newyork))
        (print-out (train-list person origin dest distance))))))

(defun generbus ()
  (dolist (person '(john mary))
    (dolist (origin '(bus-stop bus-termin))
      (dolist (dest '(downtown beach))
        (print-out (bus-list person origin dest))))))

```

D.3 Test data

D.3.1 Network files, representations and vocabulary

The initfile for DISCERN specifies the network configuration files and the general format of story representations:

```

lreps      ;;: lrepsfile
sreps      ;;: srepsfile
qasimu     ;;: qasimufile
hfmsimu    ;;: hfmsimufile
tracesimu  ;;: tracesimufile
lexsimu    ;;: lexsimufile
7 8 6 7    ;;: nslot, nword, ncase, nsent

```

The network configuration files qasimufile, hfmsimufile and lexsimufile contain the final simulation snapshot in training the FGREP modules, hierarchical feature maps and the lexicon. These files were illustrated in sections C.1.3, C.2.2 and C.3.3. The tracesimufile contains parameters for the trace feature map mechanism:

```

2 10      ;;: tracenc, tsettle
0.5 2.0 0.5 -0.1  ;;: minact,maxact,gammaexc,gammainh

```

The files lvoc and svoc contain the lexical and semantic words. The DISCERN program makes use of the fact that the mapping is one-to-one by using the same index for both lexical and semantic representations and labels. Because the semantic labels \$restaurant etc. don't have a lexical counterpart, their space is filled with foo in the lexical vocabulary:

Lexical word labels:

stories but 16 different fancy- and coffee-shop-restaurant stories. The generators produce a total of $2 \times 2 \times 2 \times 12 = 96$ different stories.

```

;;; do not generate multiple copies of the same story
(defun gener ()
  (generfancy 'big)
  (generfancy 'small)
  (genercoffee 'good 'big)
  (genercoffee 'bad 'small)
  (generfast)
; (generfast)
  (generclothing)
; (generclothing)
  (generelectronics)
; (generelectronics)
  (genergrocery)
; (genergrocery)
  (generplane)
; (generplane)
  (genertrain 'big)
  (genertrain 'small)
  (generbus)
; (generbus)
)

;;; the generx functions below generate all combinations of
;;; possible role bindings

;;; generate restaurant stories
(defun generfancy (sum)
  (dolist (person '(john mary))
    (dolist (food '(lobster steak))
      (dolist (restaurant '(mamaison leone\'s))
        (print-out (fancy-list person food restaurant sum))))))

(defun genercoffee (taste sum)
  (dolist (person '(john mary))
    (dolist (food '(spaghetti fish))
      (dolist (restaurant '(denny\'s norms))
        (print-out (coffee-list person food restaurant taste sum))))))

(defun generfast ()
  (dolist (person '(john mary))
    (dolist (food '(hamburger fries))
      (dolist (restaurant '(mcdonald\'s burgerking))
        (print-out (fast-list person food restaurant))))))

;;; generate shopping stories
(defun generclothing ()
  (dolist (person '(john mary))
    (dolist (item '(shoes hat))
      (dolist (store '(broadway bullock\'s))
        (print-out (clothing-list person item store))))))

(defun generelectronics ()
  (dolist (person '(john mary))
    (dolist (item '(cd-player tv))
      (dolist (store '(radioshack circuitcty))
        (print-out (electronics-list person item store))))))

(defun genergrocery ()
  (dolist (person '(john mary))

```



```

        if (table[i][j].value >= best)
        {
            besti2=i; bestj2=j; best=table[i][j].value;
        }
    }
    bestvalue=best;          /* value of the image unit */
    if(babbling)
    {
        sprintf(s, "->%d %d %f",besti2,bestj2,best);
        printwordout(s);
    }
}

float mem_unit_resp(indices, table, ui, uj, dim)
/* response of a single unit on the map; equation 8.1.
   The external and lateral responses are weighted
   equally, i.e. as if theta=0.5. Theta is not explicitly included
   in the code, since it can be incorporated into the sigmoid parameters. */
struct bunitdata table[maxbnets][maxbnets];
int ui,uj,dim,indices[];
{
    register int i,j, lowi, highi, lowj, highj;
    float resp=0.0, ext_resp();

    /* external input activation */
    resp = ext_resp(indices, table[ui][uj].comp, dim);

    /* lateral interaction */
    for(i=0; i<bnets; i++)
        for(j=0; j<bnets; j++)
            resp = resp + table[i][j].latweights[ui][uj] * table[i][j].prevvalue;
    return(resp);
}

float ext_resp(indices, comp, dim)
/* external input activation; equation 8.1 */
int dim,indices[];
float comp[];
{
    float seldistance();
    if(dim==0) return(0.0);
    else
        /* response is proportional to the euclidian distance of the input
           vector and the weight vector (seldistance is listed in section D.1.2) */
        return(1.0 - seldistance(indices, invector, comp, dim)/sqrt((float)dim));
}

float sigmoid(activity)
float activity;
{
    /* transform the activity to a sigmoid response between 0 and maximum;
       equation 8.2 with maxact=alpha, minact=beta */
    return(1.0/(1.0+exp(maxact*(minact-activity))));
}

```

D.2 Data generation

The test data was generated with the same templates as the training data (functions fancy-list etc., listed in section C.1.2). The word prototypes that occur in the training data were replaced by instance names, e.g. PERSON by John and Mary.

Identical stories were not repeated in the test set (multiple copies have been commented out in the function gener below). In the test data, there were 8 different fast-food-restaurant

```

    for(j=lowj; j<=highj; j++)
    {
        if (table[i][j].value < lowest)
            lowest=table[i][j].value;
        if (table[i][j].value > highest)
            highest=table[i][j].value;
    }

    /* scale the activity values within the neighborhood; equation 8.4 */
    for(i=lowi; i<=highi; i++)
        for(j=lowj; j<=highj; j++)
            table[i][j].value =
                1.0-(table[i][j].value-lowest)/(highest-lowest);

    /* create the trace on the lateral weights emanating from units
       within the neighborhood; equation 8.3 */
    for(i=lowi; i<=highi; i++)
        for(j=lowj; j<=highj; j++)
            for(ii=0; ii<bnet; ii++)
                for(jj=0; jj<bnet; jj++)
                    /* excitatory connections to units with higher activity */
                    if((ii>=lowi && ii<=highi && jj>=lowj && jj<=highj) &&
                        ((table[ii][jj].value > table[i][j].value+epsilon) ||
                         (ii == i && jj == j)))
                        table[i][j].latweights[ii][jj] = gammaexc*table[i][j].value;
                    else /* inhibitory */
                        table[i][j].latweights[ii][jj] = gammainh*table[i][j].value;
    }

    retrieve_trace(table, dim, indices)
    struct bunitdata table[maxbnet][maxbnet];
    int dim, indices[];
    {
        register int i,j,ts;
        /* for each input, start with a blank network */
        for(i=0; i<bnet; i++)
            for(j=0; j<bnet; j++)
                table[i][j].value = 0.0;

        bestvalue=1.0;
        /* settle */
        for(ts=1; (ts<=tsettle && bestvalue>=aliveact) || ts<=tsettle/2; ts++)
            /* if activity drops < aliveact after tsettle/2, the map is oscillating:
               no use continuing until tsettle */
            compute_lat_responses(table,dim,indices);
    }

    compute_lat_responses(table,dim,indices)
    /* response of the map with external input and lateral connections */
    struct bunitdata table[maxbnet][maxbnet];
    int dim,indices[];
    {
        char s[100];
        register int i,j;
        float best=(-1.0);
        float sigmoid(), mem_unit_resp();

        /* copy the current response to be the previous response */
        for(i=0; i<bnet; i++)
            for(j=0; j<bnet; j++)
                table[i][j].prevvalue = table[i][j].value;

        /* compute the new responses */
        for(i=0; i<bnet; i++)
            for(j=0; j<bnet; j++)
                {
                    table[i][j].value = sigmoid(mem_unit_resp(indices, table, i,j, dim));
                }
    }

```