ARTIFICIAL NEURAL NETWORKS AND NATURAL LANGUAGE PROCESSING

Introduction

An artificial neural network (ANN) is a type of computer that differs in important ways from the conventional computers to which the world has become accustomed. Since the mid-1980s there has been a rapid growth of interest in ANNs as an adjunct or indeed alternative to conventional computers in research areas in which computation is an essential factor, and natural language processing (NLP) is one of these. This discussion is concerned with the theory and practice of processing natural language with ANNs.

To motivate the discussion, some comments about the relevance of NLP to library and information science and about the implications of ANN technology in NLP need to be made at the outset. The essence of that relevance is simply this: given the self-evident observation that humans communicate most easily and effectively with one another using natural languages such as English, it follows that natural language is in principle the easiest and most effective way for humans to access a computer's information base. The aim of NLP research generally is to configure computers in such a way as to endow them with the facility for natural language communication. The standard way in which the task is conceptualized is as a sequence of steps something like the following:

1. The NLP device receives some physical signal as input, most often sound in spoken communication or light when reading, and, because computers cannot process sound or light, translates the signal into a form amenable to processing. This translation is referred to as *transduction*.
2. The NLP device interprets the linguistic meaning of the transduced input.
3. This meaning is related in the appropriate way to the NLP device's information base. If the input was simply declarative, then its informational content is added to the information base.
4. If, on the other hand, a response is called for, then the response is framed as linguistic meaning.
5. That meaning is the transduced into a physical output signal that a listener or a reader perceives.

At its most ambitious, such research aims to design and build the linguistic component of artificially intelligent systems that are capable of using language as humans do. The computer HAL in *2001: A Space Odyssey*—human in the sense of being motivated by human emotions and concerns and of having a perfect mastery of English, but computerlike in its nearly instantaneous access to an information base vastly superior to that of the crew members—is archetypal and, for the information scientist, the ideal. The construction of HAL-like systems has, however, proven extremely difficult and is nowhere near being achieved, but approximations to them are more tractable as a technology in library and information science applications. More particularly, NLP systems can be used as "front ends" that allow natural language access to domain-specific information processing systems. The main problem in constructing fully intelligent systems is that in order to behave like humans (at least linguistically) they would have to have access to a human's knowledge of the world as described in step (3) above, and would have to frame the meaning of inputs and outputs—that is, their semantics—in terms of that knowledge (steps (2) and (4)). This raises profound philosophical as well as practical problems of what knowledge should actually be stored and how it should be represented, and the vastness of the undertaking has so far defeated all attempts. But if one is prepared to restrict the computer's information base to a specific subject domain—a typical library catalogue database, for example—the semantics of a system's inputs and outputs are radically simplified because the nature and structure of the information is known, which in turn renders natural language input and output to and from information systems a tractable, through nontrivial, application area.

So far, we have been concerned with NLP in general. What are the advantages of using ANN over conventional technology in NLP? Research in the field is insufficiently well developed to permit a full and relatively uncontroversial answer, but there appears to be general agreement, at least among proponents of ANNs, that such advantages as ANNs offer for NLP are primarily attributable to the following differences from conventional computer technology:

- Conventional computer architecture is based on the idea of a single processor manipulating a separate memory in which the relevant data are stored, whereas ANN architecture is based on more or less numerous interconnected processors without a separate memory.
- Conventional computers are explicitly programmed to realize abstractly defined algorithms for carrying out computational tasks, whereas ANNs are not programmed but rather learn to carry out computational tasks from exposure to an environment.

A major advantage these features offer is that ANN NLP systems are more resilient in the face of unexpected or damaged input than the corresponding conventional ones. Real-world linguistic usage is often far removed from the idealizations of linguistic theory, and because conventional NLP systems are explicitly programmed, the designer has to work with an eye not only to the theory but also to all the possible corruptions—background noise, incomplete sentences, grammatically incorrect usage, and so on—that his or her system might encounter in practice. But, the world being what it is, the task of foreseeing all eventualities is difficult if not impossible, and if the system encounters something for which it has not been programmed, it will fail, perhaps catastrophically. This is known in the literature as the "brittleness" of conventional NLP systems. The corresponding ANN system, however, is resilient under the same circumstances: if the corruption is minor it will compensate and continue, if the corruption is severe it will fail, and in between it will give a continuously graded response. This is known in the literature as "graceful degradation," and can be exploited directly in library and information science NLP applications to compensate for the vagaries of user input. Rather more speculatively, ANN systems have the advantage of designing themselves by virtue of being able to learn from an environment. The designer of a conventional NLP system has to have theories that he or she can implement on a computer: a theory about the formal structure of the relevant language, a theory about the semantics of that language, and a theory of knowledge representation. But theories can be wrong. The designer of an ANN NLP system, on the other hand, need not make use of preexisting theories, but can leave it to his or her network to infer what is necessary from the environment in order for it to carry out the desired computational task. True, this is easier said than done, but research along these lines is currently in hand. One might, moreover, justly object that this "design by inductive inference" approach is superfluous for such straightforward applications as the library catalogue database referred to earlier, but it comes into its own in domains in which the structure of the information is less obvious, such as expert systems applications involving "fuzzy" reasoning.

Having briefly outlined its relevance to library and information science, we return to the theory and practice of NLP using ANNs. One possible approach would be to present a survey of recent work in ANN NLP, but given both the space constraints on this discussion and the current level of research interest in the subject, that would amount to little more than a series of brief sketches incomprehensible to anyone but the specialist. The alternative is to attempt a coherent account of some fundamental issues in NLP, and of interesting ANN approaches to them. The drawback here is, of course, that "fundamental" and "interesting" are subjective notions, and that much will be missed because it does not fit into the chosen framework. The second approach nevertheless seems preferable in terms of tractability and clarity, and is adopted here; for a recent collection of ANN NLP work with extensive further references see Ref. *1*.

The rest of this introduction defines the scope and outlines the structure of the discussion.

SCOPE

The following restrictions on the scope of this discussion are motivated mainly by the need to compress a large subject into a small space:

1. There are two broad approaches in NLP generally, which for convenience are here referred to as *cognitive NLP* and *engineering NLP*, respectively. As its name suggests, cognitive NLP is closely associated with cognitive science, a discipline that has emerged fairly recently as an amalgam of selected elements from psychology, linguistics, computer science, artificial intelligence, neurophysiology, and philosophy, and the aim of which is to develop a comprehensive theory of human cognition. Natural language has been and continues to be a primary focus because it epitomizes the class of "higher functions" in which cognitive scientists have had a particular interest. Linguistics in the Chomskyan

tradition, in particular, is concerned with developing a theory of the human mind's language faculty, and an influential view of the nature of cognition is that there is a "language of thought" analogous in terms of syntax and semantics to natural languages. Cognitive NLP bases its models on such cognitive theories and then implements them with a view to building devices that process language as humans do. Engineering NLP, on the other hand, has no commitment to cognitive theorizing. Its aim is to design devices that process natural language in relation to some specified criterion, but it is neutral on the issues of whether or not such devices process language as humans do.

Both approaches are legitimate. Choosing between them depends on what one hopes to achieve. If the primary aim is to understand and model the linguistic aspect of cognition, then cognitive NLP is appropriate. Otherwise, the obvious choice is engineering NLP, simply because one does not have to make and substantiate claims about human cognition. Cognitive NLP is by far the more ambitious for the two, and for most researchers, by far the more interesting. But it is also very much more complicated. It raises a host of difficult and often controversial philosophical, metatheoretical, and methodological issues that, in order to be understood, together presuppose at least a basic background in several of the disciplines that constitute cognitive science. One cannot hope to address these issues properly within the confines of a relatively short discussion such as this one and still deal adequately with the practicalities of current ANN NLP techniques. This discussion therefore dispenses with the cognitive dimension of NLP, and adopts an engineering NLP orientation; a useful way into the literature on the interrelationships of cognition, NLP, and ANNs are the articles in Dinsmore (*2*), which represent the mainstream of current thought on the subject and contain extensive further references.

2. Natural language communication is in spoken or written form. For any given task, processing speech is more difficult than processing text. The reason has to do with the ease with which linguistically significant input can be presented to the processing device. Text is straightforward. There is a small set of symbols—the alphabet, punctuation, space—from which words and sentences can be constructed: a word is a sequence of letters, words are clearly separated from one another by spaces, phrases are often demarcated by commas, and sentences are demarcated by full stops. As an added bonus, text is usually grammatically correct and complete; few people write consistently and grossly erroneous text or leave sentences incomplete. Representing text is moreover as simple as typing at a computer keyboard, which translates it into the equally explicit corresponding ASCII representation, which can in turn be used by the NLP processor. Speech, on the other hand, is anything but straightforward. The physical acoustic signal corresponding to written text does not consist of a few canonical elements. For any given linguistically significant element, the acoustic realization in the individual speaker will vary constantly in accordance with factors such as the acoustic context in which it occurs and the speed of articulation. Each speaker has his or her own characteristic acoustic realization, and the shape of the signal is affected to varying degrees by noise in the speaker's environment. Speech is, moreover, continuous; words are not neatly separated one from another as in text, nor are sentences consistently demarcated by, for example, pauses. And, on top of everything else, spoken language is full of ungrammatical and incomplete sentences. Much research has been and is being devoted to the question of how humans manage to extract linguistically significant regularities from so variable and frequently degraded a speech signal, but even from a purely engineering standpoint the problem is not trivial and is far from being solved. This discussion does not cover speech processing, mainly because it is an area about which I do not know enough to be able to say anything useful. It assumes text input, and concentrates on syntactic and semantic processing of natural language.
3. Various network architectures and associated learning rules have been and are being developed (*3*), and there is no hope of being able to cover them adequately here. The procedure is, rather, to choose one particular architecture that is both widely used in

ANN NLP research, and intuitively accessible. This is of course means that ANN NLP work that uses other architectures will not be covered, but that drawback is in my view balanced by greater clarity in what is, after all, an introduction to the subject.

4. The emphasis is on fairly recent research, that is, work that has appeared within the last five years or so. The earlier work and its implications have by now been digested by researchers; a brief history is available in Ref. *1*.
5. One can't know everything. Inevitably, the coverage presented here reflects the range of my reading and general competence, and cannot therefore claim to be fully comprehensive even within the limits stated above.

STRUCTURE

The processing of natural language using conventional computers (henceforth "conventional NLP") has been a research subject for about three decades, during which time a substantial corpus of theoretical and practical results has been generated. Because it brings a different technology to bear, ANN NLP can be expected to diverge more or less radically from conventional NLP, and this is already happening, as we shall see. It remains, however, that conventional NLP offers many insights into the nature of the problem, and it would be unwise of ANN NLP simply to disregard it; at the very least, ANN NLP has to show how it proposes to address fundamental issues that the conventional variety has identified, and most of current ANN NLP work does in fact make explicit or—via the conceptual framework within which it understands language—implicit reference to these issues. To understand ANN NLP and the associated research literature properly, in short, one first has to know something of conventional NLP.

Once again, scale is an obstacle; a résumé of three decades' work is simply out of the question here. It is, however, possible to convey the requisite awareness of the fundamental issues comparatively briefly. The first part of the discussion "Fundamentals of Language Processing" will do this under four headings: (1) formal language theory, (2) automata theory (3) semantic theory, and (4) representation. The rest of the discussion will then make appropriate reference to it.

Fundamentals of Language Processing

Formal languages theory provides a general definition of the notion of language that subsumes the natural languages, together with the means of generating languages with expressions having varying degrees of structural complexity. Automata theory defines abstract machines capable of processing the various structural complexity classes so generated. Semantic theory deals with how languages mean. And, finally, representation is concerned with how the abstract entities proposed by these various theories can be given a form amenable to processing by actual computers. We will look briefly at each of these.

FORMAL LANGUAGE THEORY

The standard textbook on formal language theory is that of Hopcroft and Ullman (*4*), although various others are available.

Symbols

The notion of symbol is fundamental to formal language theory (FLT). A symbol in this context is simply some physical thing that is interpretable as referring to something, which doesn't exclude much; it is standard practice to use the Roman alphabet or some subset of it. Two important things to note about symbols are

- Symbols are primitive; that is, they cannot be decomposed into constituent parts. If one defines the Roman alphabet {a...z} as the symbols to be used, then it makes no sense to decompose, say, a *d* into a loop and an ascender and then to attempt to use these two elements in subsequent language generation; "loop" and "ascender" are not part of the defined symbol set to be the lexicon of English. By that definition, English words would no longer be composed of individual letters because symbols are not decomposable; the complete physical shape *aardvark* is the symbol, not its constituent letters.
- Symbols are arbitrary relative to what they represent; anything can stand for anything else as long as everyone concerned knows the connection. Natural languages are an eloquent example of this since, excluding historical cognates and coincidence, they use different words to designate the same real-word entities: English *woman*, German *Frau*, Irish *ben.*

Strings

Having defined a set of symbols, it is possible to concatenate them so as to form sequences called *strings*. Given the symbol set {*a, b, c*}, one can form the strings *a, b, c, aa, abc, aaac, abbbbbcccc,* and so on. Intuitively, the number of strings that can be formed in this way seems very large. In fact, given only that no upper limit is placed on permissible length, an infinite number of strings can be formed from any nonempty symbol set one cares to define. Assuming a symbol set that, for convenience, we will label A, the set of all possible strings composed of the symbols in that set is designated as A*.

Languages

Given some symbol set A, a language over A* is some (itself possibly infinite) subset of A*. Assume A = the English lexicon. Then A* would include all English sentences, and a great deal of gibberish besides: *John drove his car*, but also *car drove his John, John his, his his his his his his his*, and so on. Clearly, the English language consists of some but will not all of the strings formed by concatenating the contents of the lexicon—that is, it is a subset of A*.

Grammars

The question is: how can the desired set of strings be chosen from A* so as to define a language? One possibility is simply to list them, but since subsets of an infinite set may themselves be infinite, it is not a viable one; the list would never be complete. The alternative is to give a finite specification for an infinite set, some (relatively) brief description of what characteristics a string must have in order to be admitted to the desired language. Such a specification is called a *grammar*. More particularly, a grammar gives an exhaustive list of rules for forming the strings a language; it is said to generate a language.

Grammars in FLT are *phrase structure grammars*, so called because they generate strings by building them up out of substrings called *phrases*. The notion of phrase has an intuitive basis in natural languages. In English, for example, one feels that the sequences of words that make up sentences come in clumps. Sequences such as *on the beach, behind the red car,* and *in sight* share a common pattern—preposition + optional determiner + optional adjective(s) + noun—and constitute a prepositional phrase; a typical noun phrase pattern is optional is optional determiner + optional determiner + optional adjective(s) + noun.(*horses, the man, a sad tale*). Individual words combine to form phrases, and phrases combine to form more complex phrases (i.e., noun phrase + prepositional phrase: *the man behind the red car*) or, ultimately, a complete sentence. A phrase structure grammar is essentially just a set of rules for combining words into phrases, and phrases into more complex phrases and/or sentences. Such rules are succinctly stated in the form of *productions*. For example

1. S → NP VP
2. NP → DET N
3. NP → N
4. N → man | dog | cat
5. DET → the | a
6. VP → V NP
7. VP → V
8. V → bites | catches

Here word categories and phrase types are represented by symbols (S = *sentence*, NP = *noun phrase*, and so on), and the relationships among the them by an arrow that can be read simply as "is"; the | symbol means "or." Once a set of productions has been specified, the grammar can be used to generate sentences; every sentence generation or derivation begins with the sentence symbol

S

Thereafter, the procedure is to: (1) identify the leftmost symbol in the derivation as it stands so far, (2) search the list of productions for a production in which that leftmost symbol is to the left of the arrow, and (3) rewrite the leftmost symbol in the derivation with what is to the right of the arrow in the production. So, (1) the leftmost label is currently S, (2) production 1 has S to the left of the arrow, and (3) the S in the current derivation is rewritten with what is to the right of the arrow, that is, NP VP

NP VP

Now (1) NP is the leftmost label, (2) there are two possible choices of production (2, 3) with NP to the left of the arrow; the choice is entirely free, and (3) the NP in the current derivation is rewritten by what is to the right of the arrow

DET N VP

And so on. When there are no more labels to be rewritten, the derivation is complete and the result is a sentence

the N VP

the man VP

the man bites

So far, we have been looking at phrase structure in natural language simple because it is intuitively accessible. It has, however, to be stressed that in FLT phrase structure is generalized to artificial languages such as computer programming languages and Morse code, among many other actual and potential examples. One way of looking at the string *abcdefg* is simply as a sequence of symbols: *a* followed by *b* followed by *c*, and so on. One can, however, see it as being composed of phrases, say *ab* and *cdefg*, or *abcd* and *efg; cdefg* might itself consist of subphrases *cd* and *efg*, or subphrases *c, de,* and *fg*. Which is correct? There is no intuitive guide to phrase structure in artificial languages such as there is for natural ones. The only way one can know the correct phrase structure of an artificial language string is by knowing the grammar that generated it. This last observation is crucial to understanding FLT, and to conventional NLP more particularly.

More formally, a phrase structure grammar (henceforth PSG) has three components

- A finite set of *terminal symbols*, which is the symbol set from which the strings of the language generated by the PSG are made.
- A finite set of *nonterminal symbols*, comprising the phrase type and word category labels (the NP, VP, etc. of the set of productions presented earlier).
- A finite set of productions of the form

$$x\mathrm{N}z \rightarrow xyz$$

- where

  N is a nonterminal symbol and *x, y,* and *z* are possibly empty strings consisting of terminals and/or nonterminals.

The last of these components is rather stilted. What it means is that the left side of a production may consist of any string of terminals and/or nonterminals, with the restriction that it must contain at least one nonterminal, whereas the right side can be any string of terminals and/or nonterminals (and may in fact be completely empty—the so-called null string).

This is a very general definition, but for present purposes we shall need to be more particular. By placing various sorts of additional restrictions of the above specification of how a production may look, it is possible to categorize PSGs into various types. NLP has been most interested in the following two types

Context-free Grammars

In the general definition of PSG, the only restriction on the left side of a production is that there be at least one nonterminal; in addition there can be any number of terminals and/or nonterminals. In context-free grammars (CFG), the only thing allowed on the left side is a single nonterminal. Again, the general PSG definition says that the right side of a production can be any string of terminals and/or nonterminals, including none. In a CFG, there must be something on the right; the null string is not allowed. Examples of context-free productions are

$$\begin{array}{l} S \rightarrow NP\ VP \\ NP \rightarrow DET\ N \end{array}$$

as in the foregoing example grammar; that grammar was context-free.

Regular Grammar

This is the most severely constrained type of grammar. Productions may have one of the following two forms:

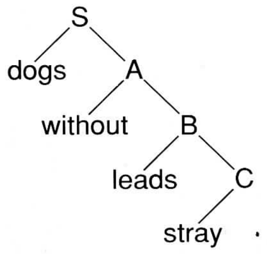
1. A single nonterminal on the left, and a single nonterminal on the right
2. A single nonterminal on the left + a single terminal on the right

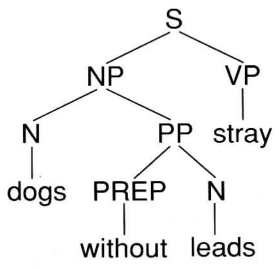
Given that grammars can be categorized into types, the question of why they should be remains. There are two reasons

- The grammatical classes form a hierarchy according to their language-generating "power", that is, according to the types of string patterns that they can generate. The least powerful class is that of the regular grammars, and next in the hierarchy are the CFGs. Linguists have long held that regular grammars cannot generate the strings that make up natural languages. Until recently many believed that the class of CFGs could, but that has now been shown to be false, and there has been a move to a yet more powerful class. For NLP, however, none of this is very important, for reasons given in Ref. (5).
- Much more important for NLP are the different kinds of phrase structures that the regular and CFGs assign the strings of the languages they generate. Consider the string *dogs without leads stray*. Here are, respectively, the regular and the CFGs that generate it

$$\begin{array}{ll} S \rightarrow \text{dogs } A & S \rightarrow NP\ VP \\ A \rightarrow \text{without } B & NP \rightarrow N\ PP \\ B \rightarrow \text{leads } C & PP \rightarrow PREP\ N \\ C \rightarrow \text{stray} & VP \rightarrow V \\ & N \rightarrow \text{dogs} \mid \text{leads} \\ & V \rightarrow \text{stray} \\ & PREP \rightarrow \text{without} \end{array}$$

Tree diagrams can be used to show the structure of the derivations according to the two grammars, again respectively





The first structure says that the string consists of *dogs* followed by the rest of the string, that the rest of the string consists of *without* followed by the rest of the string, and so on. In other words, the structure of the string is purely sequential. Moreover, given the limitations on the form of production in regular grammars, every string is purely sequential; regular grammars always assign this sequential structure and no other. The context-free structure, on the other hand, says that there is a structure over and above the purely sequential; that the first three words group against the fourth, and that in the first group *without leads* groups against *dogs*. This second analysis is the more intuitively appealing; its structuring of phrase interrelationships feels as right as the first feels wrong. And, in fact, there is overwhelming empirical support for the second. Everywhere in English sentences one finds prepositions followed by nouns followed by the (preposition + noun) pattern, and by (noun + [preposition + noun]) followed by a verb. In short, CFGs capture the phrase structure of English, whereas regular ones do not.

AUTOMATA THEORY

Automata are mathematically-defined machines that process strings generated by phrase structure grammars. Assuming some grammar G, this processing is sensitive to the structures characteristic of the strings generated by G. This section (1) describes the nature of automata, and looks at how different classes of automata actually process strings, and (2) considers a particularly important type of automation in NLP, the parser. The standard textbook is once again that of Hopcroft and Ullman. (*4*).

Automata

In describing automata, two preliminary notions have to be explained: *state* and *transition* (On these see Haugeland, (*6*). Chess is a good basis for both. At the start of a chess game the two players' pieces are arranged in rows on opposing sides of the board; the game ends when the pieces are arranged so that one player's king can no longer avoid being taken. Between beginning and end the pieces assume a sequence of configurations on the board as each player alternately makes his or her move. Each successive configuration of pieces constitutes a state—literally, the state of play. The initial opposing rows represent the start state. When a player makes the first move the state of the game changes, that is, the configuration of pieces on the board has altered. When the opposing player makes his or her first move the state changes again, and so on. In these terms, a chess game can be seen as a sequence of state changes that stops when the final state—checkmate—is reached. Now chess pieces cannot move arbitrarily around the board; each is constrained by (1) the pattern of movement prescribed for it by the rules of the game, and (2) the current state of the board. So, at any given stage of the game, the choice of possible next moves is governed by these two constraints. Or, put another way, the transitions between successive states is governed by the rules relating to the movement of types of chess pieces together with the current state of the game. A transition can therefore be regarded as the conditions under which it is possible to move from one state to another.

We consider two kinds of automata: *finite state automata* (FSA) and *pushdown automata* (PDA).

An FSA consists of the following components:

- An input tape on which the strings to be processed are written. The machine uses a read head to read one symbol at a time from the tape.
- If output is desired, an output tape on which the machine writes its output strings. The machine uses a write head to write one symbol at a time onto the tape.
- A set of symbols from which input (and output) strings are made.
- A set of states; as its name indicates, the number of states must be finite.
- A state transition function that, given the current input symbol and the current state, tells what the next state of the machine will be.

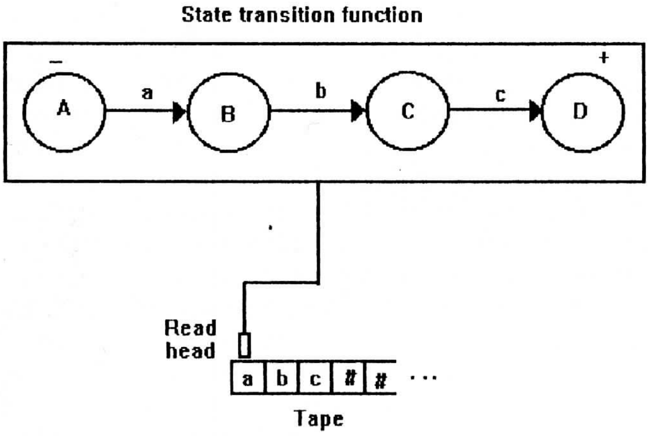
Applying this to the chess example, the set of symbols = the various types of chess pieces, the set of states = the various possible configurations of the pieces on the board, and the state transition function = the rules of chess that given the current configuration of the board and the piece currently being moved, tell what the next configuration of the board will be. What about the input string? It is the sequence of piece-moves in the course of a game; that is, since the pieces are symbols, the piece-move sequence becomes a symbol sequence—a string. Each game of chess can therefore be seen as a finite state machine processing a string.

As an example, consider an FSA to process strings generated by the regular grammar

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow bY \\ Y &\rightarrow c \end{aligned}$$

This grammar generates a language L consisting of only one string: *abc*—not very useful, but simple for illustrative purposes,(see diagram top of page 12).

This FSA is an accepter, the purpose of which is to separate strings that belong to L from those that do not. The state transition function contains four states named A–D quite arbitrarily; the transitions between states are represented by arrows; the lower case letters labeling the arrows represent current input symbols; – opposite A designates the start state, and + opposite D the final state; # is a string terminator. String processing proceeds as follows. The read head is initialized to the first slot on the input tape, and the state transition function to state A. The machine now reads one



symbol at a time from the tape, making a state transition for each one. If, when processing stops, all the input symbols have been read and the machine is in the final state D, the string is accepted as a member of L, otherwise not. The machine begins by reading the first symbol *a*, moves the head one slot to the right, and goes to state B. It then reads *b* and moves the head one place to the right, and goes to state C, and so on. Once the final state D is reached and the read head is over a string terminator, the string *abc* is accepted. Had the input string been, say, *abb*, there would have been no defined path out of state C and the machine would have stopped without reaching D, therefore rejecting the string as a member of L.

A PDA is essentially an FSA with a memory. It consists of an input and an optional output tape together with associated read and write heads, a memory called a stack, and an FSA controller that coordinates access to the tape(s) and to the stack. The tape(s) are identical to that of FSAs and require no further comment. There other two components do, however.

Stack. A stack memory is literally what its name implies: the things to be remembered are stacked one on top of the other, like a stack of books piled on the floor or the papers in an in-tray. The two crucial features of a stack memory are

- At any given time, only the top item of the stack is accessible.
- The order in which items are placed on the stack is preserved; the first is at the very bottom, the next immediately above it, and so on.

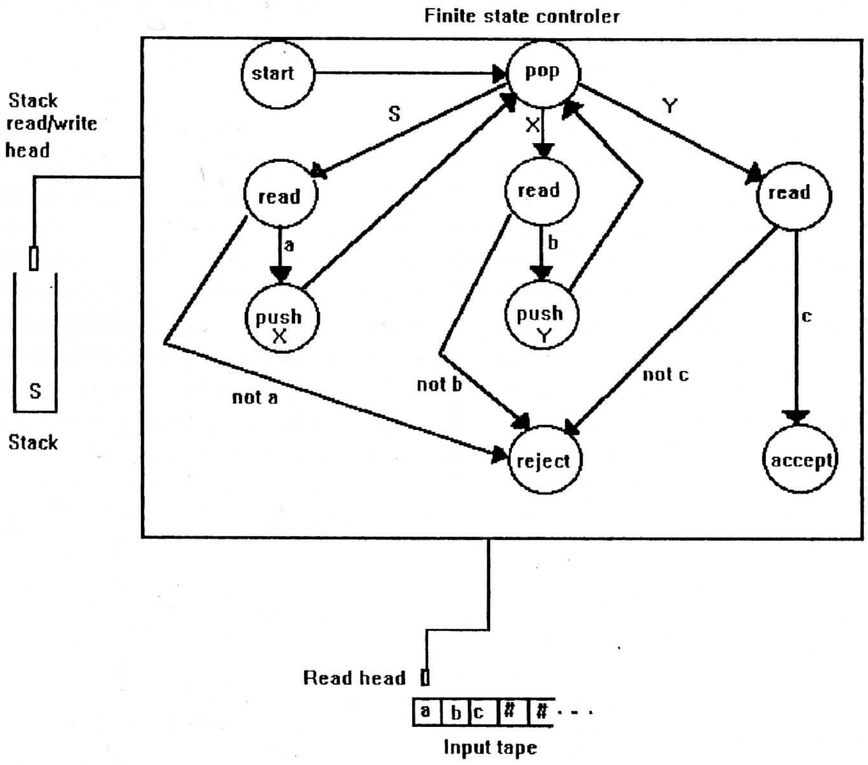
Finite State Controller. The heart of a PDA is a controller that coordinates (1) reading from and writing to the input and output tapes, and (2) putting symbols onto and taking symbols off the top of the stack. This controler is an FSA whose states are restricted to the following types:

START: Begin Processing
READ: Read a single symbol from the input tape.
WRITE: Write a single symbol to the output tape.
PUSH: Insert a single symbol onto the top of the stack.
POP: Take a single symbol off the top of the stack.

ACCEPT: Accept the input string and stop.
REJECT: Reject the input string and stop.

When appropriately configured, a finite state controller consisting of such states can process strings.

As an example, let us look at a PDA acceptor for the same language as the one used above to illustrate the operation of an FSA; the PDA is here doing exactly the same job as the FSA, but in a different way:



The PDA is initialized by putting the input tape read head over the first symbol in the string, inserting the string symbol S into the stack, and putting the machine into the START state. There is a transition to POP, and the nonterminal S comes off the stack. Follow the S-transition to the leftmost READ state. Read the terminal symbol under the read head, and move the head one place to the right; if the symbol is not *a*, then go to the REJECT state and reject the string. Otherwise, PUSH the nonterminal X onto the stack and return to POP. Now POP X out of the stack and follow the X transition to the middle READ state. Read the terminal symbol under the read head, and move the head one place to the right; if the symbol is not *b*, then go to the REJECT state and reject the string. Otherwise, PUSH the nonterminal Y onto the stack, and return to POP. Now POP Y and follow the Y transition to the rightmost READ state. Read the terminal symbol under the read head, and move the head one place to the right; if the symbol is not *c*, then go the REJECT state and reject the string, otherwise accept it.

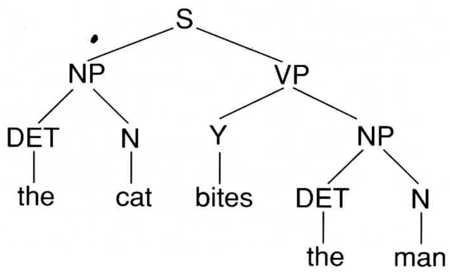
As with phrase structure grammars, the question of why different classes of automata are needed arises. The answer is that there is a direct correspondence between the classes of phrase structure grammar and of automata. Regular grammars (RG) and FSA are equivalent; for every RG, there is an FSA that will process the language it generates, and for every FSA there is an RG that generates the language it processes. The same goes for CFGs and PDAs. Note an asymmetry, though. There is no guarantee that an FSA can be found to process a context-free language, but there is a PDA for every language generated by an RG. The consequence of all this is that if one knows the grammar of the language in which one is interested, the choice of automaton class to process it is obvious.

Parsing

As we shall see in the following section on semantic theory, the assignment of meaning to a string requires that the structure of the string be known. But input strings do not come with structural descriptions conveniently attached. An NLP system therefore has to include a mechanism whereby the structures of the strings that it processes can be discovered. Such a mechanism is called a *parser*. To build a parser, one first of all has to know the grammar that generated the strings to be processed. Then, one can use an automaton of the corresponding class to construct a device that, given the grammar and a set of strings, will return for each string a structural description that shows which grammatical rules were used to derive the string, and in what order. For example, take the CFG presented earlier

S → NP VP
NP → DET N
NP → N
N → man | dog | cat
DET → the | a
VP → V NP
VP → V
V → bites | catches

The grammar generates the string *the cat bites the man*; the corresponding structure is shown by the following tree:



The function of the parser, given *the cat bites the man* as input, has to reconstruct this tree. There are standard ways of building parsers (for which see) Ref. (*7*).

SEMANTIC THEORY

So far we have been concerned with the syntactic aspect of language: how ordered symbol strings are generated and processed. Syntax is, however, not the whole story for NLP. To see what the rest is, one has to step back and ask what the point of NLP actually is. Most researchers would reply that it is to design and build computational systems that process language for some purpose—that it has to be about something. As noted in the introduction, that purpose can range from a commercially exploitable device that permits human-computer communication via natural language in, for example, a natural language front-end for a database, through to the linguistic component of an artificially intelligent system. Whatever the case, however, syntactic manipulation of uninterpreted symbols is insufficient. What is required is (1) that meaning be be derived from input strings, and (2) that that meaning be represented so that it can participate in the system's computational processes. This section consequently addresses the issue of meaning in NLP, and the next considers representation.

The discipline concerned with linguistic meaning is *semantics*. Compared to syntax, which is comparatively well understood, semantic theory is still tentative in many respects. In fact, although the question of what it means to mean has been a philosophical problem since antiquity, there is still no uncontroversial answer. Whatever meaning is, though, semanticists agree that the essence of their discipline is to associate linguistic expressions—in the terms of the current discussion, strings—with meanings. That task has various requirements; the ones most directly relevant here are

- To assign meaning to the primitive expressions of a given language—in the terms of formal language and automata theory, to terminal symbols
- To show how the meanings of these primitive expressions relate to composite expressions in the language, that is, to phrases and complete strings
- Because the whole point of language is to be able to talk about the wold around us, to show how linguistic meaning relates to the world.

These issues have been addressed in different ways by a variety of theoretical approaches to semantics. There is no hope of being able to cover even the major varieties here. Instead, the account given by *formal semantics* (also known as *Montague semantics*) is the basis for discussion (*8*); formal semantics is chosen because it has been and continues to be influential in research.

Formal semantics defines the meaning of a string as its truth conditions: a string means what the world would have to be like for it to be true. Thus, the sentence *the unicorn fights the lion* is true only if in the world in which it applies (1) unicorns and lions exist, (2) the notion of fighting between creatures exists, and (3) unicorns and lions are or can be aggressive toward one another; items 1–3 are the truth conditions of the sentence. Because truth conditions are defined relative to a world in which the language in question is to be used (a "world of discourse"), formal semantics defines a world of discourse for that language by building a model of that world. Such a model has two components

1. The entities of the world of discourse are identified—for example people, animals, objects, and ways in which these interact—and the relationship between this set of entities and the symbols of the language in question is defined. This relationship is called *denotation*, and denotation itself has two aspects.
   a. The *extension* of a symbol is the set of entities in the world model to which that symbol applies. For example, the extension of the symbol *dog* is the set of all dogs in the world.
   b. The *intension* of a symbol is rather more difficult to grasp, but intuitively it is the essence of what a symbols means—here, the intension of the symbol *dog* is dogginess in the abstract, or in other words the concept "dog". Slightly more formally, it is the specification of the conditions under which an entity can be a member of the extensional set. With the *dog* example, that specification would be a list of features such as "mammal", "four legs", "tail", "smells", and so on.
2. A specification of how the denotations of composite expressions such as phrases and strings are constructed from the denotations of the language symbols is given. This is done using the *principle of compositionality* attributed to the philosopher Frege, which says that the meaning of a composite expression is a function of the meanings of its component symbols and the way in which they are combined. Specifically, some grammar (perhaps a context-free PSG, but not necessarily) is specified to generate the strings of the language. Associated with each syntactic rule in the grammar is a corresponding semantic rule that derives the meaning of the phrase or the string from its immediate constituents; this is known as the *rule-to-rule hypothesis*. For example, if the grammar contained a syntactic rule S $\rightarrow$ NP VP, the semantic rule would specify that the meaning of a sentence is made up from the meaning of the noun phrase and the meaning of a verb phrase; the meanings of NP and VP would be similarly defined until, at the end of the string derivation, the denotations of the primitive symbols need not be derived, but are explicitly defined. To recover this meaning from an input string, an NLP system would recover its structure using a parser, and then, knowing which rules were applied in what order, would use the associated semantic rules to build up the corresponding meaning.

The assignment of denotations to linguistic symbols satisfies both the first and the third of the requirements of a semantic theory stated earlier, in that it assigns meaning to such symbols by relating them explicitly to real-world entities, and the second of the requirements is satisfied by the appear to the principle of compositionality and its implementation by means of the rule-to-rule hypothesis.

REPRESENTATION

Many problems in theoretical syntax remain, and many more in semantic theory, but the NLP researcher has one to add to them all: *representation*. The crux of the problem is this: how can the abstract entities posited in syntactic and semantic theory, together with their interrelationships, be incorporated into a real-world computational system so that they participate in the system's processing? Let us look more closely at this requirement.

A real-world computational system is physical; it computes by going through a sequence of distinct physical configurations, and each change of configuration has a physical cause. Such things as sets, grammars, automata, states, stacks, string structures, and meanings are, however, not physical but abstract. Like truth, justice, and love they do exist, but not as physical entities in the world; the notion of an abstract is a slippery one, but just to get an intuitive grasp one can—with full awareness of begging questions—think of an abstract as an interpretation of the world by humans.

Since only physical causes can affect the configuration of a real-world computational system, a way has to be found of representing abstract sets, string structures, meanings, and so on in a physical way. Once this is done, the physical representations can be made to enter into causal relationships with one another in such a way that they process physical strings, and thus constitute an NLP system. Representation of syntactic abstract entities is straightforward in principle if not always in practice. Representation of semantic abstract entities is anything but. We begin with syntax.

The entities of formal language and automata theory are mathematical, and therefore well defined. Making them computationally efficacious is as easy as writing a computer program. All high-level programming languages provide data structures for representing sets, lists, and trees, and these are adequate for representing the requisite entities. One might object that programming languages are themselves still abstract, are removed from the physical, and are consequently not truly representational. This is true, but the associated compiler translates programs into machine language, which is physical. A program implementing, say, a PDA, thereby becomes a string of physical memory register configurations that causally determine the behavior of the computer on which the program runs.

The entities of semantic theory—or, at least, of formal sematics—are also mathematical and well defined. Nevertheless, representation of them has caused and continues to cause problems for NLP. We will look at one problem of principle, and one of practice.

The problem of principle concerns denotation. Formal semantics says that the denotation of a symbol consists of an intension and an extension. Intensions are not particularly problemmatical. They were characterized earlier as sets of features that describe the essence of the meaning of symbol; that is, the criteria that define set membership, so one could use a list data structure in a programming language to, for example, list the features of *dog* that define dogginess. That is, in fact, the usual approach in conventional NLP. But it is the extensions that cause the difficulty. Formal semantics defines extensions as sets of real-world entities. Sets are mathematically well defined, but real-world entities are not, and they sit awkwardly as physical objects in a world of mathematical abstractions. This is, in my view, a serious problem for the theory, but, even putting that side, one is left with the problem of how to represent a real-world entity in a computationally efficacious form. How can human individuals be represented? By their features? No, that is intension. Pictures? Smells? To my knowledge, no good solution exists; this problem is, in fact, a major argument against the possibility of ever endowing a computer with true semantics—that is, with an "internal" understanding of the relationship between the strings it processes and the real world (*9*, *10*). The problem, in essence, is that without some way of representing real-world entities in a computer, it cannot have knowledge of the world, and hence cannot have true understanding of language, but that leads on to things beyond the scope of this discussion.

The practical problem is one of scale. Even assuming that a way could be found of representing real-world entities in a computational system, and thereby of implementing a formal semantic model in such a system, one would have to ask (1) how many such entities there are, and (2) what the rules governing their interrelationships are. For toy experimental worlds with relatively few entities and simple interrelationships, a formal semantic NLP model seems tractable (assuming a solution to the extension problem), but for NLP the world of discourse is our own real world, and both the number of entities and their interrelationships are vast and terrifyingly complex; it may be possible to represent the world in a computer, but many (myself included) doubt that it can. Allied to this is the self-evident observation that, unlike the well-defined model worlds of formal semantics, the real world is constantly changing and unpredictable, and it is difficult to see how a representation of a predefined model in a computational system could keep track of it.

ANN NLP

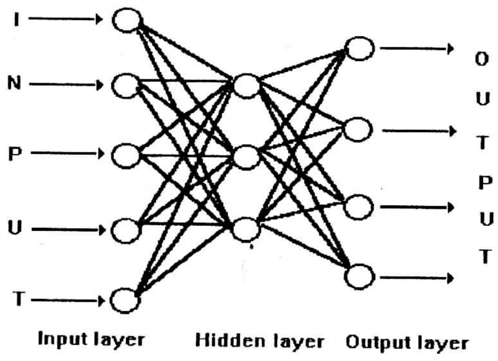
Having described some of the fundamental ideas and issues in NLP, we are now in a position to see how ANN research relates to them. There is no question of being able to give a one-for-one account, that is, to counterpose well-developed ANN-based theories of language, language processing, semantics, and representation to the conventional ones. ANN NLP has been intensively studied for less than a decade, and the corresponding theories are still being developed. Much of this development is being done in terms of how the conventional (so-called classical) paradigm in cognitive science relates to the ANN-based ("connectionist") one, and coherent syntheses are beginning to emerge, but this whole area was excluded from the scope of the present discussion not because it is peripheral to NLP—it is directly relevant—but because it would extend the discussion beyond reasonable limits. Given its engineering NLP orientation, what follows concentrates on some of the main"practical" results achieved so far.

The discussion is in two parts. The first describes how a popular variety of ANN looks and works, and the second goes on to consider how this type of ANN is used in NLP. The second part is itself divided into three subsections: (1) syntatic processing, (2) representation of structure, and (3) representation of lexical meaning.

ANN COMPUTATION

An ANN usually consists of numerous interconnected processors. It computes by receiving inputs on a set of input processors, propagating these inputs along the connections with the other processors, and delivering the result on a set of output processors; often a set of processors that has no direct connection with the outside world is involved in the propagation of inputs to outputs. The nature of the computation performed is determined solely by the pattern of connectivity among processors.

This is a highly general and rather stark description of what an ANN is and how it computes, and it will need to be elucidated in what follows. It subsumes a variety of ANN architectures that have been and are currently being developed (*3*) and, as noted in the introduction, what follows cannot hope to cover them all. The procedure is, therefore, to choose one particular architecture that is both widely used in ANN research, and accessible at an introductory level. That architecture is the *distributed*, *feedforward*, *three-layer net* with the associated *back propagation* learning rule.

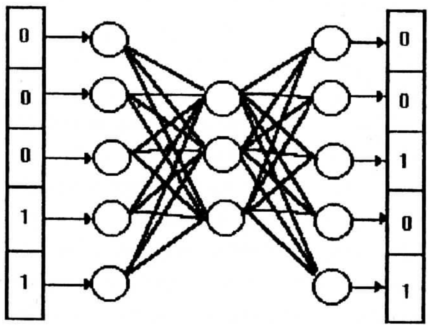


Circles here stand for processors, henceforth called *units*, and the lines between them for connections between units. The units are arranged in layers according to their function: those that receive inputs from the net's environment, those that deliver the result of the net's computation to the environment, and those with no direct connections to the environment ("hidden units") whose role is to take part in the computation. All units in the input layer are corrected to all those in the hidden layer, and all those in the hidden layer to all in the output; there are no direct input-output connections.

To see how such a net computes, one has to understand the nature of (1) the inputs and outputs, (2) the connections, (3) the units, and (4) the interaction between units and connections.

Inputs and Outputs

A net takes vectors as inputs, and delivers vectors as outputs; vectors are simply lists of numbers. Thus, if one wanted a net to do arithmetic, one could represent numbers not by the usual arabic numbers but in binary form: 0 = 00000, 1 = 00001, 2 = 00010, 3 = 00011, and so on. Each such binary representation can be regarded as a vector: a list of 0s and 1s. So, if one wanted a net to take an integer as input and return its double as output, the input and output vectors would look like the following:



Note that there is one input unit for each component of the input vector, and one output unit for each output vector component.

Connections

It has been noted that input is propagated through the net along the connections among units, and that by the time this propagation reaches the output layer a computation has been performed. How the computation happens has not yet been described. It is, however, the case that the connections are crucial to the computational process. Specifically, the connections can take on varying strengths. Indeed, if the net is to carry out any interesting computation, the strengths of the interconnections among its constituent units must have significant variation, since the configuration of connection strengths is what determines the computation that the net is capable of performing. How a net comes to acquire the relevant connection strengths is something to which we shall be returning.

Units

The first of the above diagrams differentiates units according to their roles. In principle, however, all units are identical as processors, that is, in terms of the functions they compute. This is a slight oversimplification in that input units differ in one important respect from the hidden and output units, but that difference can be dealt with in a moment. Each unit performs a very simple computation: it sums all the inputs coming into it over its incoming connections, applies some function to the sum, and then sends the result along all its outgoing connections. The precise nature of the function applied to the sum is not particularly important here; its purpose is to ensure that the sum is mapped into some well-defined interval, say 0..1 or -1..1, so that what gets propagated along the outgoing connections in response to all the inputs, no matter how many or how large or small, is a value in that interval, say 0.67 or -0.32. The input units differ simply in this: each one only gets a single input from its environment, and that value is sent directly along the outgoing connections without any function being applied to it.

Interaction Between Units and Connections

Take any unit u_i in the input layer. (For ease of reference, units are usually numbered from top to bottom from 1..*n*, and referred to by number; the *i* subscript here stands of any given unit.) Assume some value in the corresponding component of the input vector v_i, say 0.74. The input unit propagates the value 0.74 along all its outgoing connections. But we have seen that connection strengths in any interesting net will vary significantly. The outgoing connections from u_i will therefore exhibit some degree of variation. This means that the input value 0.74 will propagate more or less strongly along each connection. If one represents strength by real values in the range 0..1, and the five connections coming out of u_i are 0.32, 0.27, 0.89, 0.97, 0.12, then the value that actually reaches the hidden units along these connections will be

(0.74 x 0.32), (0.74 × 0.27), (0.74 × 0.89), and so on. This applies to all input units. It also applies to all the hidden and output units. A given hidden unit receives values from the input units in accordance with the strength of its connection with each of them. A function is then applied so as to reduce the sum of these values to an interval, say 0..1, and that a reduced value is then propagated along each of the hidden unit's outgoing connections. But these connections vary, too, so that the reduced value is propagated more or less strongly to the output units. And finally, each output unit sums all its inputs, applies the function (which is usually called a "squashing" function because it squashes sums into some interval), and outputs the result into the corresponding output vector.

Putting all this together, one can see how an ANN computes. Assume well-defined input and output vector sets, and a net whose connections are such as to enable it to carry out some mapping between the two. Now take an input vector and apply it to the input layer. The units propagate the input values along their outgoing connections, the value sent along each connection being adjusted in line with the connection's strength. These adjusted values arrive at the hidden layer. Each hidden unit sums the values on its incoming connections, applies the squashing function, and sends the result on its outgoing connections; these values are in their turn adjusted in line with the connections between hidden and output layers. The adjusted values arrive at the output layer, at which each unit sums the values on its incoming connections, applies the squashing function, and outputs the result of the corresponding component of the output vector. The output vector is then the result of the computation. A computation in an ANN can therefore be seen as a wave of values sweeping through the net from input to output layers, and being modified by the connection strengths as it does so. Note, finally, that all this happens in parallel. All the components of the input vector are presented simultaneously, and all input units propagate the values independently and simultaneously; each hidden layer unit can proceed with its summation, squashing, and value propagation as soon as all its input values are in, without having to wait for any of the others. The same goes for the output units.

Clearly, the connection strengths in a net are crucial to its computational capability. Suitable connection strengths have so far been assumed, but how are connections appropriate to some specific computation arrived at? In very small nets they can be handcrafted, and for some kinds of nets it is possible to calculate them explicitly, but in general connection strength configurations in an ANN are learned. The learning mechanism described here is *back propagation*, one of many such mechanisms currently available. Assume a well-defined input-output mapping, that is, a set of pairs in which the first element is the input and the second the associated output. The mapping suggested above, in which the input is an integer and the output is that integer doubled, is a good example; the function computed is "double the input". Back propagation uses such pair-sets to train ANN's, and it works like this. Take a pair—for example, binary 3 and its double, binary 6, as in the above diagram. Present the first element of the pair as input, and let it propagate through to the output layer. Now compare the result at the output layer to the target output—that is, what the net should be associating with the input: the second element of the pair which here is the binary vector representing 6. If the actual result on the output layer corresponds exactly or closely within some tolerance, do nothing, because the connection strengths happen to be exactly those required to perform the desired mapping, that is, to associate 3 with 6. If, however, there is a discrepancy, adjust the connections in such a way that if the input were presented again, the actual result on the output layer would be closer to the target output. Now present the same input again, let it propagate through the newly modified connections to the output, and compare it once again to the target. If there is still a discrepancy, adjust the connections and try again. This procedure is repeated until the actual and target outputs correspond. When they do, the connections have been configured so as to carry out the desired 3-6 mapping. But the function "double the input" should apply to any integer, or at least to some useful finite range of integers. Training a net to handle such a finite set of associations involves presenting it with each pair in the set and following the learning procedure just described. Given a sufficiently large net, back propagation can find a single set of connection strengths to carry out multiple pair associations, thus allowing the net to compute interesting functions. Details of how back propagation actually works are found in Ref. *11*.

LANGUAGE PROCESSING

Syntactic Processing

Language processing is intrinsically a sequential business. In spoken English, for example, words are uttered in temporal sequence, and the hearer processes them in that sequence; in written English the sequence is spatial, from left to right, on some surface. ANNs for language processing consequently have to be able to deal with symbol sequences.

The first step is to find a way of representing symbol sequences so as to be amenable to processing by ANNs. In automata theory, machines process symbol strings directly: they can take the physical shape *a* and process it, then *c*, and so on. But there is no obvious way for a net with the architecture described above to work directly with *a* or *c*, or for that matter with *cat* or *dog*. The symbols have to be suitably encoded. This is standard procedure even in conventional NLP, although the encoding is mostly hidden from the programmer, and is generally forgotten about. A conventional NLP program written in a high-level language refers to *a* and *c*, and to character strings *cat* and *dog*, but in fact the compiler represents each letter by means of some standard encoding (typically ASCII). ASCII encoding simply sets alphabetic characters in a one-to-one relationship with the binary numbers 0..127. For example, 00100000 = space character, 01100001 = *a*, 01100010 = *b*, and so on. We have already seen that such binary numbers can be construed as vectors and input directly into an ANN; a character string would thereby simply be a sequence of binary vectors, one for each character, that the net could process one at a time. Thus the string *abcdefg* would be encoded as ACSII for *a* followed by ASCII for *b*, followed by ASCII for *c*, and so on. This extends readily to NLP, as the following diagram shows:

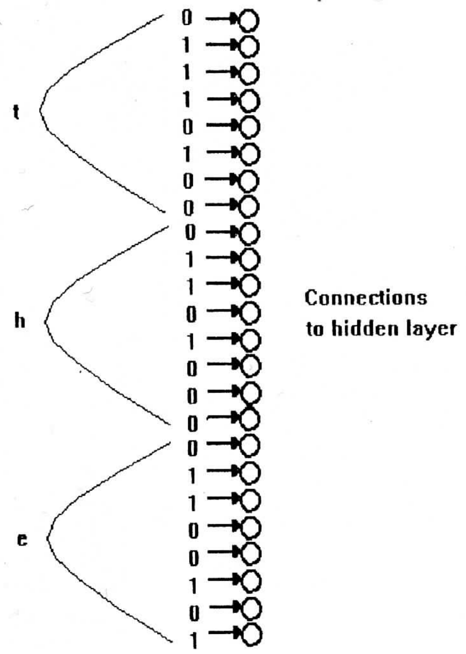
t	h	e		m	a	n	
01110100	01101000	01100101		01101101	01100001	01101110	

....

To enable a net to input one word at a time, two steps are necessary

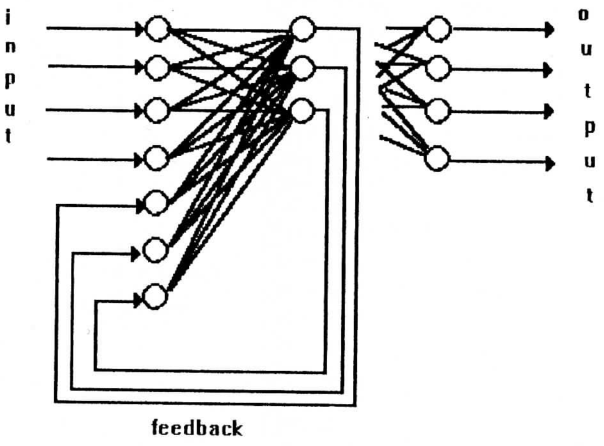
- Concatenate the binary vectors for each of the constituent letters, forming one long vector.
- Give the input layer of the net exactly as many units as the concatenated word vector has components, and input the word vector directly.

Thus for the word *the*



Various other encoding schemes are possible. The one described here, for example, is one of several in which there is a purely arbitrary relationship between the symbol and the encoding: binary for *c* is very different from binary for *o*, even though they are much closer to one another in terms of physical shape than, say, *c* and *m*. An encoding scheme that preserves and exploits such similarities can have some considerable advantages for ANN NLP (see, e.g., Ref. 5).

Having proposed an approach to representing symbol sequences amenable to ANN processing, the next step is to find a way of allowing ANNs to process them. The net architecture described above is not self-evidently suited to the task, since computation is a matter of associating (vector representation of a) single symbol with the (vector representation of) another. True, a symbol sequence could be handled by presenting each symbol to the net in turn, but that would not solve the problem, since each successive computation would be independent, and would have nothing to do with those that preceded. But everything that has been said about language processing so far depends on relating the component symbols of a string to one another in a structured way—the symbols do have something to do with one another, and the processing mechanism must reflect this. The most widely adopted solution to this problem is to augment the above ANN architecture by making it *recurrent*. To see what his means, we will look at the intuitively simplest kind of recurrent net, aptly enough called a *simple recurrent network* (SRN), first proposed by Jordan (*12*) and subsequently developed by Elman (*13*, *14*); SRNs are the basis for a good deal of recent ANN language processing research. An SRN looks like the following (connections between hidden and output units have been truncated for clarity):



This augments the architecture described earlier in two ways. First, a number of units equal to the number of hidden layer units has been added to the input layer. Second, there are feedback lines from the hidden units to those that have been added to the input. Why is this architecture better for sequential processing of strings? Because, if it is run at discrete time steps so that the configuration of the hidden layer at time $t - 1$ is jointly involved with the input at time t in generating the next hidden layer configuration and associated output, then it is a finite state machine.

- The set of input and output vectors is the symbol set.
- The set of hidden layer configurations is the state set.
- The connections between input and hidden layers are the next state function in that for every combination of current input symbol and current state (i.e., the hidden layer configuration fed back to the input layer) they generate a characteristic associated next state in the hidden layer.
- The connections between hidden and output layers are the output function, in that for every combination of hidden units they generate a characteristic associated output in the output layer.

Because it is an FSA, a trained SRN processes strings exactly as an FSA does. An FSA begins in an initial state; the SRN's hidden layer is initialized to a known configuration. An FSA then reads the first symbol, and this symbol, together with the current state, causes the machine to move to the next state (and deliver any output); in that new state it reads the next symbol, and that symbol, together with the current state, causes the machine to move to the next state (perhaps with output), and so on to the end of the string. The SRN reads the first symbol vector into the external input part of the input layer, and has the initial state in the state part of the input layer. That combination is propagated to the hidden layer, which assumes a new configuration, which in turn generates output. This new configuration is fed back to the input layer. The net now reads the next symbol vector, and the cycle is repeated for the rest of the string.

We will now look at examples of research that uses the SRN architecture and developments of it for language processing. That research can be divided into two broad categories, here referred to as top-down and bottom-up. Assume the existence of some language whose strings are to be processed. Faced with this task, the top-down approach works within the framework for language processing provided by formal language and automata theory. It proceeds by specifying a grammar that generates the string set, designing an automaton capable of carrying out the desired processing, and then uses ANNs as a technology to implement the automaton. The bottom-up approach, on the other hand, does not begin with any prespecified theory. Rather, it exposes an ANN to the string set, and on the basis of syntactic regularities in the string set, expects the net to configure itself by inductive inference into a device capable of carrying out the required computation. The net thus configured can (but need not) be understood in terms of automata theory, and a grammar for the string set can subsequently be derived from the automaton if desired. The examples that follow should clarify all this.

The top-down approach is exemplified by Moisl (*5*); I cite my own work first not out of self-aggrandizement, but because this is its natural place in the structure of the discussion. The aim is to design a parser for general NLP, and then to implement it in an ANN. The argument proceeds in the following stages.

First, the deterministic pushdown transducer (DPDT), a restriction on the general class of pushdown automata, was proposed as an adequate formal model for general NLP. Essentially, (1) an adequate model must be capable of assigning syntactic structural descriptions to all possible NL strings, (2) the structural descriptions must be empirically justifiable, and (3) the structural descriptions must support a nontrivial compositional semantics, since, as noted earlier, the whole point of a parser is to support semantic interpretation of strings. The DPDT is the simplest adequate class of automata for items (2) and (3); given a maximum length restriction on strings, it satisfies item (1) as well. Since NLP is about the design and construction of physical devices processing physical strings, such a restriction is not an obstacle: all real-world natural language strings are finite in length and in fact very short relative to the unbounded-length strings postulated in computational theory and linguistics.

Second, given the string length restriction, the DPDT can be simulated by a finite state transducer (FST). "Simulation" in this context means "black box equivalence." Given a device A whose inner workings are known, and another B whose inner workings are unknown (i.e., a black box), if the input-output behavior of B is exactly the same as that of A, then B simulates A. In other words, they compute the same function, and in terms of the job they do they are equivalent, although they do it in ways that are possibly different. Here, the DPDT and the FST compute the same function in that for any input string both return the same structural description. They do so in different ways, however, because of their different architectures.

Third, an SRN was trained to perform the same input-output mapping as the DPDT using back propagation, thus simulating the DPDT.

Fourth, a computer implementation was used to test the performance of an SRN trained to simulate the operation of a DPDT on a set of natural language strings. The following characteristics emerged:

- The SRN successfully simulated the DPDT in that it was able to correctly parse all the strings in the training set.
- In addition the SRN was able to parse certain strings generated by the underlying grammar on which it had not been specifically trained. This indicates that training an SRN on a subset of the language generated by the grammar will in general be sufficient to enable it to parse all the strings generated by it.
- The SRN failed to parse strings that were grossly ungrammatical (i.e., misplaced or missing constituents), just as the DPDT did, and as both should: automata by definition process only the strings generated by the associated grammar. It was, however, able to parse what might be termed "corrupt" strings; that is, strings in which there were relatively minor lapses in grammaticality. These were:
  - —Certain spelling errors in which individual letters were wrong or missing. For example, with respect to the string *the lark flies*, the SRN was able to restore the correct form from *th lark flies*, *the lerk flies*, *the lard flies* (but not *the Irk flies*) and complete the parse correctly.
  - —In some but not all cases, the net was able to compensate for subject-verb number disagreement and complete the parse. For **the bard chant the holy verse*, for example, the SRN correctly restored *the bards chant the holy verse* and returned the correct structural description.

  The DPDT, in the other hand, failed when confronted with either kind of corruption.

Finally, the main conclusion was that the proposed approach offers what NLP work has traditionally set as a major goal: devices whose behavior is consistent with abstract linguistic theory (competence), and at the same time displays resilience in the face of degraded input that characterizes real-world linguistic environments (performance). In conventional NLP systems competence is relatively easy to achieve, but the combination with performance is much more difficult; in the research literature, conventional systems are described as "brittle" because of the difficulty they have in handling degraded real-world input. In the ANN implementation of a conventional model proposed here, the performance aspect comes without any extra effort; it is a by-product of the fact that the implementation medium is an ANN.

Another example of the top-down approach to ANN NLP is the work of Williams and Zipser (*15*), whose trained a recurrent net to implement the finite-state controller of a Turing machine (the most powerful class of automata), and Touretzky (*16*), who developed a (nonrecursive) ANN that creates and manipulates such conventional data structures as stacks and trees.

The bottom-up approach is exemplified in the work of several individuals and research groups. This approach addresses a general problem that has been and continues to be an issue in nonconnectionist research: the language acquisition/grammatical inference problem, stated by Pollack (*17*) as follows. [See Lucas (*18*) for a recent collection of papers on this subject].

In its narrowest formulation, it is a version of the inductive inference or 'theory from data' problem for syntax: discover a compact mathematical description of string acceptability (which generalizes) from a finite presentation of examples. In its broadest formulation it involves accounting for the psychological facts of native language acquisition by human children, or even the acquisition of language itself by homo sapiens through natural selection.

Most of the work currently being done in ANN grammmatical inference is concerned not with natural but with artificial languages, the aim being to establish theoretical results about ANN language processing. But, as we saw, natural languages are a special case of language as defined in formal language and automata theory, and these results are consequently of direct relevance to NLP.

Several research groups have been looking at the "easiest" kind of grammatical inference: given a set of strings, let the net infer a finite state automaton capable of processing them, and of generalizing to previously unseen strings generated by the corresponding regular grammar.

Servan-Schreiber et al. (*19*, *20*) "show that the SRN can learn to mimic closely a finite state automaton . . . In particular, we show that it can learn to process an infinite corpus of strings based on experience with a finite set of training exemplars." The procedure was to generate a large (60,000) set of binary strings of lengths ranging from 3 to 30 digits, using a regular grammar, as the training set for the net. For each string, each successive digit was given to the net as input, and the target output was the next digit in the string. Once training was complete, the net was tested as follows:

- It was presented with a large number (70,000) of purely random binary strings, a few of which happened to be grammatically legal relative to the grammar that generated the original training set, but most of which were not. It was able to separate the grammatical from the ungrammatical ones perfectly.
- It was presented with 20,000 grammatical strings, some of them from the training set but others not, and was able to process them all.
- It was presented with grammatical strings much longer than any on which the net was trained—more than 100 digits—and was able to process them all.

The conclusion was that by exposure to a linguistic training environment, the SRN was able to infer a finite state accepter for strings generated by the regular grammar underlying the training sample; implicit here is that it had learned to process not only the training sample, but to generalize to all grammatically legal strings. The chief theoretical result, therefore, is that exposure to a linguistic training environment induced in the SRN an FSA corresponding to a regular grammar on the basis of a relatively restricted sample string set.

Servan-Schreiber et al. analyzed the network with a view to determining how it managed to work as an FSA. This was done by saving all the hidden layer configurations that the net assumed during the text phase, and then applying cluster analysis to the whole set of such configurations. Cluster analysis is a mathematical technique for determining the relative closeness of vectors to one another on some metric of similarity. The result was that the hidden layer configurations were not spread randomly throughout the available vector space, but that they clustered into groups that are interpretable as the states of a finite state automaton. In the course of an analysis, moreover, an additional property with important implications for NLP was discovered. When the individual state vector clusters were examined, it was discovered that the hidden layer configurations making up a given cluster were differentiated according to the path traversed before the node is reached, which means that as the net goes through state transitions, it preserves a memory of the state trajectory. As the authors point out, this is not a characteristic of conventional finite state automata, which by definition have no memory of previous states. The implication for ANN language processing, and for NLP in particular, is that so called long-distance dependencies can be efficiently handled by SRN FSAs. An example of such a dependency in natural language is subject-verb number agreement: *The boys who came over the bridge the other day have gone home/The boy who came over the bridge the other day has gone home.* Conventional FSAs can handle such dependencies, but require much duplication of states, and are very inefficient for string sets of any useful size. SRN implementations of FSAs do it by "shading" states; that is, by developing clusters of very similar vectors, each of which is essentially a single state, but in which the small differences among vectors preserve state trajectory information. Moreover, they do it for free, as a by-product of grammatical inference.

In subsequent work Servan-Schreiber et al. developed aspects of the above, and in particular the following:

- Refined the interpretation of hidden-layer clustering: "The finite state machine that the net implements can be said to approximate the idealization of an FSA corresponding exactly to the grammar underlying the examplars on which it has been trained. This SRN, given sufficient resources... converges on the theoretical FSA in the sense that it can be used as a perfect finite state recognizer for the strings generated by the corresponding grammar." However, "the internal representations do not correspond to that idealization" in the sense that the states of an SRN FSA are not primitive, but consist of closely related state vectors that can as a group be interpreted as states of a conventional FSA, but that also carry state trajectory information.
- Extended the study of state trajectory information encoded by SRN FSAs from languages generated by deterministic to languages generated by nondeterministic FSAs. When, for a given current input and current state, the theoretical FSA allows two or more possible next digits, the SRN learns to activate both the corresponding states simultaneously.
- Examined in greater detail how state trajectory information is learned by the net, and the factors that influence the net's ability to learn it. These factors are
  1. The nature of the state trajectories; some can be learned and some cannot.
  2. The size of the net; if it is too small, the state trajectory information that learned initially is progressively eliminated as training proceeds.
  3. String length: as the distance between dependencies increases, the net finds it correspondingly difficult to maintain trajectory information.

In their conclusion, Servan-Schreiber et al. stressed the importance of the SRN's ability to preserve state trajectory information, and thus to process long-distance dependencies, to NLP: "The ability to exploit long distance dependencies is an inherent aspect of human language processing capabilities, and lies at the heart of the general belief that a recursive computational machine [*note:* a conventional formal grammar/automaton] is necessary for processing natural language. The experiments we have done with SRNs suggest another possibility: it may be that long distance dependencies can be processed by machines that are simpler than fully recursive machines, as long as they make use of [state] path information."

The work of Servan-Schreiber et al. has been described in some detail. Examples of similar work by other researchers can therefore be dealt with more briefly. Very closely related is that of Cleeremans et al. (*21*) and Das and Das (*22*); others use recurrent network architectures to achieve inductive inference of finite state automata from formal language string sets, but differ from Servan-Schreiber et al. mainly in the following respects:

- SRNs are conceptually the easiest recurrent network architecture to grasp. Other researchers use more complex architectures: Giles et al. (*23–25*), Smith and Zipser (*26*), Pollack (*17*), and Watrous and Kuhn (*27*).
- Network training procedures used by the researchers just mentioned differ in various ways that are not particularly important for the purposes of this discussion.
- The dominant tendency in ANN research generally is to use tabula rasa learning: choose a network architecture, randomize the initial hidden layer and the connections, and hope that the net will infer regularities from the learning environment ab initio. There is no reason why this should be so, however, and some researchers have begun to study the effect of starting with a net that has some "knowledge" of the environment's regularities before training begins—in practice, whose connections at the onset of learning have been appropriately pretuned—and concluded that learning is not only quicker and more reliable, but in some cases allows the net to learn things that it failed to learn ab initio. [See Sharkey and Skarkey (*28*) for a recent discussion with further references; see examples in Omlin and Giles (*29*), Giles and Omlin (*30*)].
- There has been some work on grammatical inference of automata belonging to the more powerful class of PDAs using techniques similar to those we are currently looking at. See Sun (*31*) for discussion and further references.
- Pollack (*17*) adopts a physicist's view of ANNs as nonlinear dynamical systems that can be trained to approximate the string processing capabilities not only of finite state but also of more powerful classes of automata by virtue of their fractal dynamics, and thereby offers an alternative to automata theory as an interpretative framework for analysis of ANN language processing devices.

The final work we shall be considered in this syntax section is that of Elman (*13*, *14*), who uses SRNs to infer FSAs not from formal string sets, but from sets of strings such as one would find in natural language. It is described at some length both because of its intrinsic interest for ANN NLP, and because it has been very influential in research. Elman describes a series of experiments of increasing complexity.

- A letter string was constructed to see if the net could infer any of its regularities. This was done in three steps. First, the consonants b, d, g were randomly assembled into a 1000-letter string. Then each of the letters in the string was replaced according to the rules

$$\begin{aligned} &b \rightarrow ba \\ &d \rightarrow dii \\ &g \rightarrow guuu \end{aligned}$$

  Finally, each of the consonants and vowels was encoded by a 6-bit binary vector, in which each of the bits corresponded to a phonological/phonetic feature; b, for example, was 101001 (1 = consonant, 0 = vowel, 1 = stop, 0 = high, 0 = back, 1 = voiced). Note that the net has no way of knowing what the coding means. The net was presented with the resulting vector string such that, for each input vector, it had to learn to predict the next one. After training, the net was tested by a vector sequence constructed in the same way, but with a different initial randomization of b, d, g. For each input vector, predict the next one. The most obvious result was that, in general, network error was high and the prediction wrong where the next letter was a consonant, and the network error was low and the prediction correct where the next letter was a vowel. This shows that the net had inferred such regularities as there were to infer in the string: consonants were random, and therefore unstructured in their pattern of occurrence in the string, but once a consonant was encountered, the vowel sequences were systematic, and the net was able to learn them.

- The next experiment extended the one just described to see if the net could extract the notion "word" from letter sequences. Here, instead of the previous arbitrary letter sequences, actual natural language strings were used. Specifically, 200 strings between 4 and 9 words long were concatenated, forming a stream of 1270 words, corresonding to 4963 letters. Each letter was then assigned a distinct 5-bit vector encoding. Input was one letter at a time, and the task was to predict the next letter. The result was that network error for the first letter of a word was high, because the order of words from so small a sample of strings is not very predictable, but the error declined for the rest of the letters in the word. The net inferred the predictable regularities in the strings, and they corresponded to the conventional notion of *word* as certain kinds of letter sequences.
- The next step was to see if an SRN was able to infer any syntactic regularities with the general approach used in the preceding two experiments. The first step was to attempt this with minimally complex strings: two and three words on the patterns [subject-verb] and [subject-verb-object]. 10,000 such strings were generated using a restricted vocabularly of 29 words. These strings were, moreover, not arbitrarily constituted on the above patterns, but obeyed standard syntactic and semantic natural language restrictions. Thus transitive verbs took objects but intransitive ones did not; a verb like *eat* could only take something edible (*cookie* but not *rock*) as an object, and so on. Finally, all the strings were concatenated into a single sequence, each word was assigned a characteristic but arbitrary 31-bit binary vector code, and, as before, the net was given one vector at a time as input and asked to predict the next one. Now, in interpreting the results, it has to be kept in mind that while natural language strings in general and the training strings in particular have structure, word sequence is not deterministic; noun followed by verb is a standard pattern, but one can have *men see*, *men run*, *men eat*, and so forth. The net could, therefore, not be expected to predict the next word even after training, and it did not do so. What it did do, given some word, was learn what words could follow it, and activate the output layer accordingly; that is, generate an output vector that was a combination of the vector codes for each of the words in question. For example, after *rock* one expects words like *move* and *break* vectors.

  To see how the net did this, cluster analysis was applied to the set of internal layer configurations obtained by presenting the complete training sequence once to the trained net. The clusters showed that the net had partitioned the words from the training strings into the basic noun and verb categories, and then subcategorized these in accordance with grammatical and semantic features. Thus hidden layer configurations for animate and inanimate nouns formed separate clusters, the animate noun configurations subcategorized according to human and nonhuman, and so on. Thus the net was able to infer from the training strings (1) lexical categories and subcategories, and (2) the ways in which these categories and subcategories could combine syntactically.
- Finally, there was an experiment to see what regularities an SRN could infer from more complex string structures than those used in the immediately preceding one. This included a recursive structure—multiply embedded relative clauses like *dog who chases cat sees girl*, and even the classic center-embedding *boys who girls who dogs chase see hear*, which has a long history in linguistics—and the requirement that subject-verb number agreement be maintained across embeddings (*dog who boys feed sees girl*). Encoding and training were analogous to those in the preceding experiment, and are not described again. The main results were as follows:

—Again, word sequence is not deterministic, and the net's behavior was as before: it predicted all the words that could follow a given one in the output layer.
—The net predicts that a singular subject will be followed by a singular verb, and a plural subject by a plural one, or else by the relative marker who, which is correct.
—The net correctly predicts word order in embedded structures, and maintains subject-verb number agreement across them; that is, it has found a way of representing a long-distance dependency. This was observed also by Servan-Schreiber et al.

In addition, Elman's work bears on other relevant issues. For one thing, he suggests an approach to preconditioning of nets for learning tasks that is different from that of Giles and Omlin referred to earlier. His approach is incremental training by means of increasingly complex training data. Elman observes that when an attempt was made to train the net with the full range of string structures, including ones with embedded clauses, it was unable to learn the task. He consequently trained the net in stages. In the first stage only simple sentences were used. When training with those was complete, a degree of domain-specific "knowledge" had been built into the net, and it was now possible to introduce more complex strings into the training set incrementally as further training proceeded, and in this way the net was able to learn the task it had been unable to learn before.

Representation of Structure

In 1988 two prominent cognitive scientists, J. Fodor and Z. Pylyshyn, published a now famous article (*32*) in which they denied that connectionism—a blanket term for ANN-oriented research—was a viable alternative to conventional (or as they called it, "classical") language and automata theory as a paradigm for cognitive science, and indeed that connectionism had little relevance at all for cognitive science, except perhaps as an implementation medium. Its tone was not calculated to mollify connectionists, and there ensured a spirited and still current debate that, polemics aside, has been beneficial in that fundamental issues have had to be raised and discussed. [See Dinsmore (*2*) for current state of debate; also Clark (*33*).] The central issue was the representational capacity of ANNs; as such, their critique is directly relevant to present concerns. The essence of their argument was: (1) that ANNs are quintessentially finite state devices, and as such that the only structure they can represent is sequential ordering—that is, right (or alternatively, left) branching trees; (2) that to model cognition in general and human language processing in particular, one needs a much richer diversity of structures than finite state sequencing; and (3) that ANNs are consequently inadequate by nature for modeling human cognitive abilities. Connectionists generally accept the need for a diversity of structures, and so since 1988 the search has been on to find ways in which ANNs might represent them, and thus to refute Fodor and Pylyshyn's arguments. We will look at what is in my view the most promising approach: *superpositional representation*.

To understand superpositional representation, it is necessary to reexamine the concept of compositionality already mentioned in connection with semantic theory. The key work here is that of Van Gelder (*34*; also Van Gelder and Port, *35*; critique in Christiansen and Chater, *36*), on which what follows is based. Van Gelder begins by extending the notion of compositionality beyond semantics: "In the most general (and vague) sense, any item is said to have a compositional structure when it is built up, in a systematic way, out of regular parts drawn from a certain determinate set; those parts are then components or constituents of the item." He then proceeds to focus this idea. As a first step, general conditions that any compositional scheme must satisfy are proposed.

- There is a set of primitive types (symbols, words, etc.) P_i; for each type, there is available an unbounded number of instances or tokens.
- There is a (possibly unbounded) set of expression types R_j; likewise, for each type there is available an unbounded number of tokens.
- There is a set of transitive and nonreflexive constituency relations over these primitive expression types.

Before going on with the argument, some comments on "transitive and nonreflexive constituency relations," and on the type/token distinction, are in order. As regards the first, *transitive* and *nonreflexive* can be ignored for present purposes; a *constituency relation* is simply a specification of how parts in a compositional scheme can relate to one another, as in the production rules of the grammars we looked at earlier. A type is an abstraction corresponding to the notion of intension in semantic theory, while a token is a physical example—a member of the extension—of a type; there is the concept "dog" (type) and there are actual dogs (tokens).

Now, the above conditions do not constitute a compositional scheme, but only state the properties that any such proposed scheme must have. If one accepts these conditions, then any actual compositional scheme one cares to propose has to do two distinct things. First, the scheme has to be abstractly defined; that is, the primitive symbol and expression types together with their constituency relations have to be stated. Second, the representation of the abstract scheme has to be specified; that is, how the token of the primitive symbol and expression types are going to be physically represented. These two things are standardly collapsed into a single specification. Consider a context-free PSG for a small fragment of English as an example of a compositional scheme.

S → NP VP
NP → DET NOM
NOM → N PP
NOM → N
PP → PREP NP
VP → V NP
DET → the | a
N → man | woman | cat | dog
V → sees | bites | kicks | walks
PREP → in | on | with

The type and token specifications are as follows:

- The primitive symbol types are to be English words, and the corresponding tokens are to be printed representations of English words. It may initially seem strange to regard words as abstracts, since they are so familiar in daily use. The common word for a household feline can be represented as marks on a page (CAT), as acoustic waves when spoken, or as the electronic register configurations that a computer assumes when the word is typed, among many other possibilities. The physical instantiations of the word are very different, but a single thing underlies them all: the abstraction, the concept, the word.

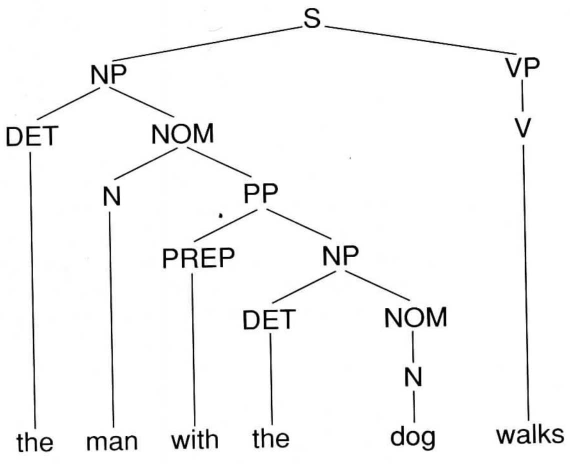
- The expression types of English are to be such things as prepositions, noun phrases, and so on, and the corresponding tokens are to be instantiated as uppercase printed labels. The abstractness of notions such as "preposition" and "noun phrase" is manifest; there are no noun phrases in the physical world.
- The abstract constituency relations are rules that specify that word categories and phrase types can combine in various ways; noun phrases are followed by verb phrases, a prepositional phrase consists of a preposition followed by a noun phrase, and so on. The corresponding representation is a set of production rules consisting of tokens of word and expression types. Constituency is represented by writing an expression token on the left, and some combination of word and expression tokens on the right. The order of constituents is indicated by *spatial concatenation* of tokens on the right side; noun phrase followed by verb phrase is physically represented by printing NP and then VP from left to right; *spatial concatenation* is italicized because it is important in what follows.

Have separated out abstract and representational specification in a compositional scheme, it can reasonably be supposed that, for any given abstract specification, there is in principle more than one way to represent it. Specifically, van Gelder proposes *temporal superposition* as an alternative to spatial concatenation as a way of representing ordering of constituents. (For further information on superposition see Van Gelder, *37*.) To get an intuition of the difference, consider two representations of the sentence *the man with the dog walks*, which is generated by the above example CFG. The printed one just given represents the sentence by placing one word after another on a page; the words are spatially concatenated. But the sentence can also be represented in speech, in which case the acoustic realization of one word follows another not in space but in time; it is *temporally superposed*. (Note that they are not temporally *concatenated*. Spoken words to not hang in the ether so that once the utterance is complete they are all somehow available simultaneously. Rather, each successive word is uttered, then fades away and is replaced by the next.) Temporal superposition offers ANN research a way out of the representational inadequacy with which Fodor and Pylyshyn have charged them. To see how, consider again the example sentence just given. Its structure (according to the CFG that generated it) can be represented as a tree diagram (see diagram top of page 34).

Alternatively, it can be represented as a bracketed expression.

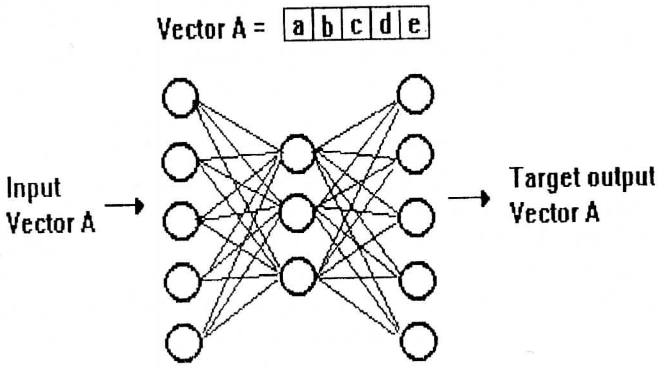
S (NP(DET(*the*)NOM(N(*man*)PP(PREP(*with*)NP
(DET(*the*)NOM(N(*dog*))))VP(V(*walks*)))

In both representations of this complex structure, (1) the constituent tokens are explicitly present, and (2) their relationship to one another in the structure is represented by spatial configuration. This is precisely the kind of representation that is needed to support nontrivial compositional semantics, and to which ANNs are unsuited, as Fodor and Pylyshyn have argued. ANNs are, however, capable of

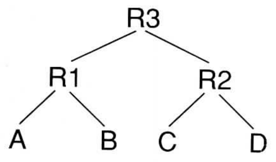


representing this kind of structure by temporal superposition, an example of which we will now consider.

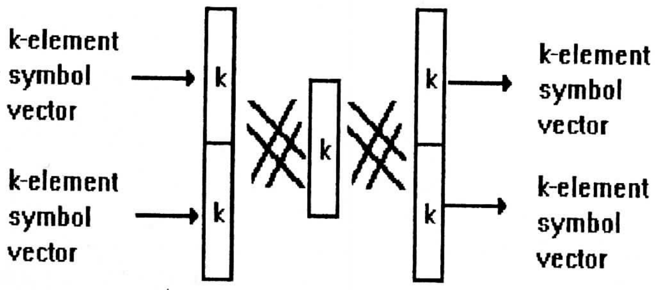
Pollack (*38*, *39*) devised an elegant solution for representing complex tree structures in the hidden layer of a nonrecurrent ANN exactly like the one described earlier, which he calls *recursive autoassociative memory* (RAAM). It depends on using such a net as an autoassociator, which is simply a net in which the input and the target output vectors are identical—the input is associated with itself.



Given an input vector and an identical target vector, back propagation is used to train the net so that when the input vector is presented, it is reproduced on the output layer. What is the point of this? Back propagation has the property that, for some I/O vector pairs, it automatically discovers a hidden layer configuration that encodes the association. Here, because the input and output vectors are identical, and because the number of hidden units is smaller than the number of I/O units, the configuration that back propagation discovers for a given pair is a compressed encoding of the I/O vector. Now, let us take a simple tree structure and see how it can be encoded in the hidden layer of an autoassociator.



Let each of the nonterminal symbols A, B, C, D be represented by a vector having, say, *k* elements. Now take a net with 2*k* input units, 2*k* output units, and *k* hidden units.



The net is trained as follows:

- As input, present the A-vector to the top *k* input units, and the B-vector to the bottom *k* units. As target output, use the A-vector for the top *k* output units, and the B-vector for the bottom *k* units. In other words, autoassociate the terminals A and B of the tree to be represented. Apply back propagation, and when training is complete take the hidden layer configuration, label it R1, and save it for later use.
- Do exactly the same for C and D. Label the hidden layer configuration R2, and save it for later use.
- Now take R1 and R2 and autoassociate them. When training is complete, label the hidden layer configuration R3, and save it.

The claim is that R3 encodes the tree. How is this claim justified?

The trained net can function both as an encoder and as a decoder for the tree. To encode, only the input and hidden layer are used. First input the AB vector pair; R1 is generated in the hidden layer, and is saved. Then the CD vector pair is input, and R2 is generated in the hidden layer; it too is saved. Finally, R1 and R2 are input, and R3 is generated and saved. The claim is that R3 encodes the tree. If that is so, then it should be possible to reconstruct the tree from R3. That is in fact possible. Take only the hidden and output layers, and input the R3 that was previously saved. R1 and R2 are generated in the output layer, and saved. Now input R1, and the AB vectors are generated in the output. Finally, input R2, and the CD vectors are generated in the output. Since the means for encoding the tree in, and decoding the tree from, the hidden layer are provided by the dynamics of the net, it can be concluded that the hidden layer represents the tree.

Note, however, the two ways in which this ANN representation differs from the one using spatial concatenation.

- Spatial concatenation has all the constituent tokens physically present in the representation. The ANN representation does not. If one were visually to examine R1, R2, or R3 in the hope of finding the A, B, C, and D vectors, one would be disappointed. A, B, C, and D are not spatially there, but encoded in the R-vectors and recovered by the dynamics of the net, that is, by the operation of the net over time.
- The representation based on spatial concatenation represents the constituency relations of the tree by deploying the tokens in space. The ANN representation represents the constituency relations temporally; first AB is encoded into R1, then CD into R2, and finally R1R2 into R3. Thus the term *superpositional representation*; the constituents are literally superimposed on one another in the hidden layer, and are only recoverable by the network dynamics.

Superpositional representation is, therefore, a viable alternative to spatially concatenative representation. Spatially concatenative representational schemes are standard and therefore familiar, but there is no intrinsic reason to prefer them. Superpositional representation is natural to ANNs, and opens the way to the representation of complex structure necessary in NLP. In particular, it supports what Van Gelder calls "functional compositionality", in which compositionality is achieved not by a spatially concatenated structure but by a representation manipulated by a process: "We have functional compositionality where there are general, effective, and reliable processes for (a) producing an expression given its constituents, and (b) decomposing the expression back into those constituents."

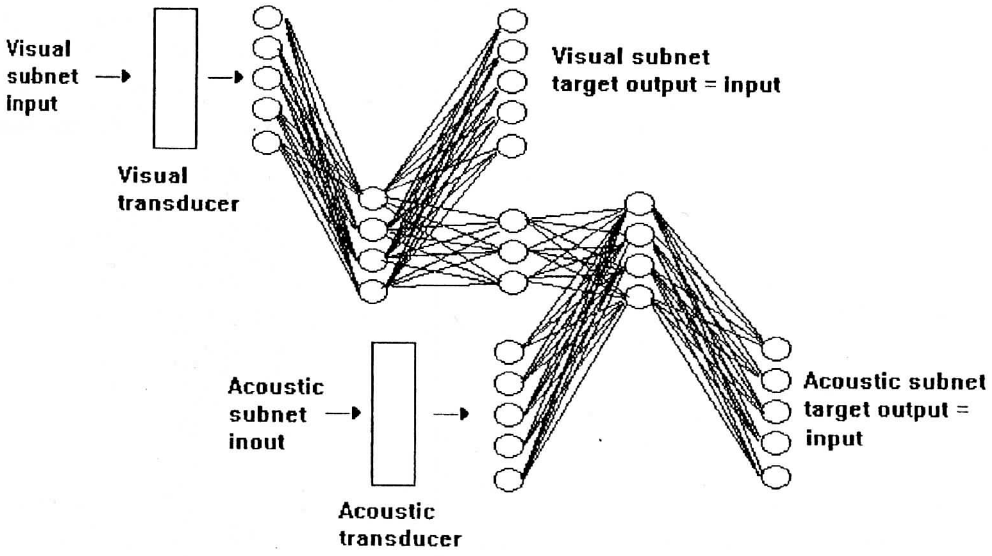
Pollack shows how a RAAM can encode much more complex tree structures than the one just discussed, and furthermore can encode a multiplicity of trees. Other researchers have, moreover, developed RAAMs along the lines discussed above (Sharkey, *40*; Blank, Meeden, and Marshall, *41*); and proposed variants on superpositional representation, chief among these being Smolensky's tensor product representation, *42–44*; discussion in Van Gelder *37*).

Representation of Lexical Meaning

In formal semantics the meaning of an expression is a function of the denotations of its constituent primitive symbols, and of the way in which they are structured. We have seen that ANNs have the capability of representing complex structures; it remains to be considered whether and how denotations might be represented and associated with linguistic symbols in ANNs.

In the conventional approach to constructing an NLP system with compositional semantics, the system designer instantiates a formal semantic model, and in particular specifies a language together with the world in which the semantics of that language is to apply. This means that the designer both has to supply his system with all it knows about that world—the range of entities in it, their characteristics and properties, how they interact with one another—and also to associate the words of the language in question with these entities and their interrelationships. The system lacks any direct contact with the world; it is, so to speak, blind, deaf, incapable of tactile sensation, and entirely reliant on the designer's description of the world for its semantics. By contrsat, the ANN approach described here does not require the designer to specify a language and a world of discourse. Instead, he or she places the system into a real-world environment, provides it with a range of perceptual mechanisms, and relies on the learning capability of ANNs to (1) inductively infer the primitive symbols/words and the syntax of the language in its environment, (2) inductively infer significant non-linguistic regularities from the environment that will serve as the denotations for the words of the language, and (3) associate words with their denotations. In other words, the designer specifies a network architecture, provides an environment, and lets the system learn its own syntax and semantics.

Let us look at a very simple example of how an ANN might learn to associate a word with its denotation. Note that the aim here is to convey the essence of what is involved; this example is not derived from any particular published research, nor is it intended as a proposal for a workable system. Specifically, we want the net to learn a representation of the word *table*, and to associate it with a denotation; that is, with an intension and an extension. The architecture is as follows: for clarity, far fewer units than would actually be required are pictured:



The visual transducer is a device that takes light as input, and delivers a vector encoding of it as output; it can be thought of as a suitably modified videorecorder. Similarly, the acoustic transducer takes sound as input, and outputs a vector encoding; it can be thought of as an enhanced microphone. How the translation into vectors takes place is not crucial for the present purposes. The network itself consists of two autoassociator subnets, one for visual and one for acoustic input, and a third subnet that links the other two so that the hidden layer of the visual subnet is its input layer, and the hidden layer of the acoustic subnet is its output layer.

We assume as an environment a real-world room containing the above net. Various tables are placed into the center of the room, one at a time, and the net's visual transducer is focused on the place at which each successive table stands. The tables differ to varying degrees: big, small, round, square, rectangular, wood, chrome/plastic, ornate, simple, and so on. Once any given table is in place, the word *table* is uttered; on each such occasion, the net goes through a learning cycle. For learning, the following three things happen simultaneously:

- The visual transducer is activated. The resulting vector is input to the visual subnet, which autoassociates it and develops a compressed representation of the input in its hidden layer, as described previously.
- The acoustic transducer is activated. The resulting vector is input to the acoustic subnet, which autoassociates it and develops a compressed representation of the input in its hidden layer.
- The linking subnet associates the compressed representation of the visual input with the compressed representation of the acoustic input, and represents the association in its hidden layer.

This cycle is repeated until the net is fully trained. Whether or not it is fully trained is determined by a simple test. Every so often, bring the sequence of tables into the room and activate the net, but do not utter *table*. For each table, the vector representation of *table* should appear in the output units of the acoustic subnet. If not, continue the training.

Once fully trained, the net is tested. Because is is trained, we already know that presentation of any of the training set tables will elicit the vector encoding of *table*. But there are two more possibilities.

- What happens if some other kind of entity in the room—a vase, a mirror—is presented to the net? The answer is that because the net has not been trained to associate that entity with anything, the output will be indeterminate.
- What happens if a table that is not part of the training set, and that differs from all the training tables to some degree, is presented? The answer here is that the output will be the vector encoding of *table*. Why? It is a general property of this type of ANN architecture that if a net is trained to associate a set of mutually similar input vectors with a single target output, it will adjust its connections in such a way that all the inputs generate the same hidden layer configuration. If a vector that is not one of the input training vectors but that is nevertheless similar to them is presented to the trained net, then that vector will generate the target output even though the net had never been explicitly trained to make this association. In other words, the net generalizes to novel inputs on the basis of its experience. Now, the vectors representing tables in our room constitute such a mutually similar input set, simply because from a purely visual point of view tables have certain invariant features—a surface a more or less standard distance from the floor, legs, a particular orientation to the room (not on the walls or ceiling), and so on—and these features are captured by the visual part of our example net. They are all, moreover, mapped onto a single output, *table*. Thus, shown a new table, the net uses its existing "knowledge" of tables and generalizes to it.

How does all this relate to ANNs' ability to represent denotations of symbols? Let us take extension and intension separately.

The extension of a word in formal semantics is the set of real-world entites that the word designates. In the current world of discourse—the table-room—the extension of *table* is the set of tables on which the net was trained. Presentation of any member of the extension set as input elicits the transducer's encoding of *table* as output. Since that encoding could be restored as the original acoustic signal using a transducer that goes from vectors to sound waves, we can say that the net associates the set of tables with the word *table*. If, moreover, any other entity in the room is presented—a vase, a mirror—the output is indeterminate, as noted. It follows that the net has learned to represent the association of *table* with and only with its extension.

The intension of a word in formal semantics is a rule for determining membership in the extension set. A representation of a word's intension in an NLP system requires that the system, given any arbitrary real-world entity as input, must be able to decide whether that entity belongs to the word's extension. We have just seen that the net rejects any nontable input, and to this extent represents the intension of *table*. But is is only to an extent. What if a table is presented that the net has never encountered before, and that differs from those in the training set? We have also seen that the net accepts the new table by assigning it to the extension of *table*. The net has, in short, learned to represent the rule for determining membership in the extension set of *table*, and can therefore be said to have learned its intension.

There are numerous problems, both philosophical and practical, with this approach to the representation of lexical meaning, and development of it has only begun. For recent work within this general paradigm see Harnad (*10*), Plunkett et al. (*45*), and Dorffner (*46*); for a critique see Christiansen and Chater (*36*).

Prospects

Conventional NLP is decisively top-down in the sense that it begins with syntactic and semantic theories, and then implements (some aspect of) a given theory in a conventional computer. One approach to ANN-based NLP is to regard ANNs as an alternative implementation medium. As we saw, this can have advantages over conventional computer implementation in terms of resilience in the face of degraded real-world input. As such, the ANN-as-implementation-medium approach has an immediate technological application in the construction of NLP systems. In the longer term, however, it seems clear—at least to me—that ANN NLP research will increasingly be interested in bottom-up inference of linguistic knowledge from the real-world environment. This means that future ANN NLP systems will be components of larger systems that also include ANN-based sensory input/output components, and with which they will interact (Pfeifer and Verschure, *47*; Peschl, *48*). The theoretical framework for understanding such systems will, moreover, move away from formal language and automata theory, and toward dynamical systems and complex systems theory (Pollack, *17*), but that is another story.

REFERENCES

1. R. Reilly and N. Sharkey, *Connectionist Approaches to Natural Language Processing,* Lawrence Erlbaum Associates, Hillsdale, N.J., 1992.
2. J. Dinsmore, ed, *The Symbolic and Connectionist Paradigms: Closing the Gap,* Lawrence Erlbaum Associates, Hillsdale, N.J., 1992.
3. J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation: Santa Fe Institute Studies in the Sciences of Complexity,* Addison-Wesley, Redwood City, Calif., 1991.
4. J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Redwood City, Calif., 1979.
5. H. Moisl, "Connectionist Finite State Natural Language Processing." *Connec. Sci.,* **4,** 67–91 (1992).

6. J. Haugeland, *Artificial Intelligence: The Very Idea,* MIT Press, Cambridge, Mass., 1985.
7. A. Aho and J. Ullman, *The Theory of Parsing, Translating, and Compiling,* Prentice-Hall, New York, 1972.
8. R. Cann, *Formal Semantics: An Introduction,* Cambridge Textbooks in Linguistics, Cambridge University Press, Cambridge, UK, 1993.
9. R. Searle, "Minds, Brains, and Programs." *Behav. and Brain Sc.,* **3,** 417–457 (1980).
10. S. Harnad, "The Symbol Grounding Problem." *Physica D,* **42,** 335–346 (1990).
11. D. Rumelhart and J. McClelland, *Parallel Distributed Processing,* 2 vols., MIT Press, Cambridge, Mass., 1986.
12. M. Jordan, "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine." *Proceedings of the Eighth Annual Conference of the Cognitive Science Society,* 1986.
13. J. Elman, "Finding Structure in Time." *Cog. Sc.,* **14,** 179–211 (1990).
14. J. Elman, "Distributed Representations, Simple Recurrent Networks, and Grammatical Structure." *Machine Learn.,* **7,** 195–225 (1991).
15. R. Williams and D. Zipser, "Experimental Analysis of the Real-Time Recurrent Learning Algorithm." *Connec. Sci.,* **1,** 87–110 (1989).
16. D. Touretzky, "BoltzCONS: Dynamic Symbol Structures in a Connectionist Network." *AI,* **46,** 5–46 (1990).
17. J. Pollack, "The Induction of Dynamical Recognizers." *Machine Learn,* **7,** 227–252 (1991).
18. S. Lucas, "Grammatical Inference: Theory, Applications, and Alternatives." *IEE Colloquium,* London, 1993.
19. D. Servan-Schreiber, A. Cleeremans, and J. McClelland, "Learning Sequential Structure in Simple Recurrent Networks," in *Advances in Neural Information Processing Systems,* 1, D. Touretzky, ed. Morgan Kaufmann, San Mateo, Calif., 1989.
20. D. Servan-Schreiber, A. Cleeremans, and J. McClelland, "Graded State Machines: The Representation of Temporal Contingencies in Simple Recurrent Networks." *Machine Learn,* **7,** 161–193 (1991).
21. A. Cleeremans, D. Servan-Schreiber, and J. McClelland, "Finite State Automata and Simple Recurrent Networks." *Neural Computa.,* **1,** 372–381 (1989).
22. S. Das and R. Das, "Induction of Discrete-State Machine by Stabilizing a Simple Recurrent Network Using Clustering." *Computer Sci. Informatics,* **21,** 35–40 (1991).
23. C. Giles, G. Sun, and H. Chen, "Higher Order Recurrent Networks and Grammatical Inference," in *Advances in Neural Information Processing Systems 2,* D. Touretzky, ed. 1990.
24. C. Giles, G. Sun, and H. Chen, "Grammatical Inference Using Second-Order Recurrent Neural Networks." *Proceedings of the International Joint Conference on Neural Networks, IJCNN 91,* Morgan Kaufmann, San Mateo, CA, 1991.
25. C. Giles and C. Miller, "Extracting and Learning an Unknown Grammar with Recurrent Neural Networks," in *Advances in Neural Information Processing Systems 4,* J. Moody, S. Hanson, and R. Lippmann, eds. Morgan Kaufmann, San Mateo, Calif., 1992.
26. A. Smith and D. Zipser, "Learning Sequential Structure with the Real-Time Recurrent Learning Algorithm." *Internat. J. Neural Syst.,* **1,** 125–31 (1989).
27. R. Watrous and G. Kuhn, "Induction of Finite State Languages Using Second-Order Recurrent Networks." *Neural Computat.,* **4,** 406–414 (1992).
28. A. Sharkey and N. Sharkey, "Connectionism and Natural Language," in *Grammatical Inference: Theory, Applications, and Alternatives,* S. Lucas, ed. IEE Colloquium, IEE, London, 1993.
29. C. Omlin and C. Giles, "Training Second-Order Recurrent Neural Networks Using Hints," in *Machine Learning: Proceedings of the Ninth International Conference,* D. Sleeman and P. Edwards, eds. Morgan Kaufmann, San Mateo, CA, 1993.
30. C. Giles and C. Omlin, "Rule Refinement with Recurrent Neural Networks." *International Joint Conference on Neural Networks, IJCNN 93,* Morgan Kaufmann, San Mateo, CA, 1993.
31. G. Sun, "Learning Context-free Grammar with Enhanced Neural Network Pushdown Automaton," in *Grammatical Inference: Theory, Applications, and Alternatives,* S. Lucas, ed. IEE Colloquium, IEE, London, 1993.
32. J. Fodor and Z. Pylyshyn, "Connectionism and Cognitive Science: A Critical Analysis." *Cog.,* **28,** 3–71 (1988).
33. A. Clark, "The Presence of a Symbol." *Connec. Sci.,* **4,** 193–206 (1992).

34. T. Van Gelder, "Compositionality: A Connectionist Variation on a Classocial Theme." *Cog. Sci.,* **14,** 355–384 (1990).
35. T. Van Gelder and R. Port, "Beyond Symbolic: Prolegomena to a Kama–Sutra of Compositionality," in *Symbol Processing and Connectionist Models in Artificial Intelligence and Cognition: Steps Toward Integration,* V. Honavar and L. Uhr., eds. 1993.
36. M. Christiansen and N. Chater, "Connectionism, Learning, Meaning." *Connec. Sci.,* **4,** 227–252 (1992).
37. T. Van Gelder, "Defining 'distributed representation.'" *Connec. Sci.,* **4,** 175–192 (1992).
38. J. Pollack, "'Implications of Recursive Distributed Representations," in *Advances in Neural Information Processing Systems 1,* D. Touretzky, ed. Morgan Kaufmann, San Mateo, Calif., 1989.
39. J. Pollack, "Recursive Distributed Representations." *AI,* **46,** 77–105 (1990).
40. N. Sharkey, "The Ghost in the Hybrid: A Study of Uniquely Connectionist Representations." *AI Simu. Behav. Q.,* **79,** 10–16 (1992).
41. D. Blank, L. Meeden, and J. Marshall, "Exploring the Symbolic/Subsymbolic Paradigm: A Case Study of RAAM," in *The Symbolic and Connectionist Paradigms: Closing the Gap,* J. Dinsmore, ed. Lawrence Erlbaum, Hillsdale, N.J., 1992.
42. P. Smolensky, "Tensor Product Variable Binding and the Representation of Symbolic Structures in Connectionist Systems." *AI,* **46,** 159–216 (1990).
43. P. Smolensky, "Connectionism, Constituency, and the Language of Thought," in *Meaning and the Mind: Fodor and His Critics,* B. Loewer and G. Rey, eds. 1991.
44. P. Smolensky, G. Legendre, and Y. Miytata, *Principles for an Integrated Connectionist/Symbolic Theory of Higher Cognition,* technical report CU-CS-600-92, Computer Science Department, University of Colorado at Boulder, 1992.
45. K. Plunkett, C. Sinha, and M. Moller, "Symbol Grounding or the Emergence of Symbols? Vocabularly Growth in Children and a Connectionist Net." *Connec. Sci.,* **4,** 293–312 (1992).
46. G. Dorffner, "A Step Toward Subsymbolic Language Models Without Linguistic Representations," in *Connectionist Approaches to Natural Language Processing,* R. Reilly and N. Sharkey, eds. Lawrence Erlbaum, Hillsdale N.J., 1992.
47. R. Pfeifer and P. Verschure, "Beyond Rationalism: Symbols, Patterns, and Behaviour." *Connec. Sci.,* **4,** 313–326 (1992).
48. M. Peschl, "Construction, Representation, and the Embodiment of Knowledge, Meaning, and Symbols in Neural Structures." *Connec. Sci.,* **4,** 327–338 (1992).

BIBLIOGRAPHY

This bibliography lists some basic readings in the various aspects of ANN NLP touched on in the preceding discussion.

Formal Language and Automata Theory

Denning, P., Dennis, J., and Qualitz, J., *Machines, Languages, and Computation,* Prentice-Hall, 1978.
Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley.

Formal Methods and Natural Language

Cann, R., *Formal Semantics: An Introduction,* Cambridge Textbooks in Linguistics, Cambridge University Press, Cambridge, UK, 1993.
Partee, B., ter Meulen, A., and Wall, R., *Mathematical Methods in Linguistics,* Kluwer, 1990.

Conventional NLP

Allen, J., *Natural Language Understanding,* Benjamin/Cummings, 1987.
Gazdar, G. and Mellish, C., *Natural Language Processing in LISP,* Addison-Wesley, 1989.
Smith, G., *Computers and Human Language,* Oxford University Press, 1991.

Artificial Neural Networks

Hertz, J., Krogh, A., and Palmer, R., *Introduction to the Theory of Neural Computation: Santa Fe Institute Studies in the Sciences of Complexity,* Addison-Wesley, 1991.

Rumelhart, D. and McClelland, J., *Parallel Distributed Processing,* 2 vols., MIT Press, 1986.

ANNs and Cognitive Science

Bechtel, W. and Abrahamsen, A., *Connectionism and the Mind: An Introduction to Parallel Distributed Processing in Networks,* Basil Blackwell, 1991.

Clark, A., *Microcognition: Philosophy, Cognitive Science, and Parallel Distributed Processing,* MIT Press, 1989.

Dinsmore, J., ed., *The Symbolic and Connectionist Paradigms: Closing the Gap,* Lawrence Erlbaum Associates, 1992.

ANN Natural Language Processing

Reilly, R. and Sharkey, N., *Connectionist Approaches to Natural Language Processing,* Lawrence Erlbaum Associates, 1992.

HERMANN MOISL