

## 3 Connectionist Foundations

---

*If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them.*

-Henry David Thoreau

Cognitive models fall into two broad categories: the *symbolic* and the *subsymbolic*. It is difficult to clearly define the two categories; each is usually described by its tendencies rather than any single definitive property. Symbolic processing is characterized by hard-coded, explicit rules operating on discrete, static tokens. Subsymbolic processing is associated with learned, fuzzy constraints affecting continuous, distributed representations. These two categories roughly define the ends of a spectrum of models.

The Analogator project is firmly rooted in the subsymbolic part of the spectrum; Analogator is implemented as a connectionist network operating on distributed representations. The goal of this chapter is to provide background knowledge of connectionism sufficient to understand the basic terms and operations of general subsymbolic computation. Specifically, this chapter examines motivations for using connectionism, learning via back-propagation, hidden layer activations and analysis, simple recurrent networks, and tensor product algebra.

### 3.1 Why connectionism?

My foremost reason for choosing connectionism as the basis for a cognitive model is its potential for learning and generalization. Adaptive connectionist networks are often capable of detecting statistical regularities in the set of input patterns presented to them. After being suitably trained, connectionist networks are able to perform in reasonable ways when exposed to novel input patterns based on what they have learned during training. This ability, often called *generalization*, *induction*, or *interpolation*, allows a network to operate much more flexibly than a system that relies on explicit, rigid ‘rules’.

Connectionist networks also generally exhibit graceful degradation of their performance as a task increases in difficulty or in the presence of noise. This is in contrast to symbolic systems, which typically cannot handle noise, and have clear processing limits. Generally, symbolic systems either work, or they do not—there is no middle ground. Connectionist networks also operate in a parallel fashion, so processing can be quite fast once learning has been completed.

However, there are some major disadvantages to connectionism. First, there is no guarantee that a network will be able to learn a given task. In addition, even if the network is able to find a solution, it might take a very long time to reach an acceptable

level of performance. Connectionist networks typically learn by gradually adjusting their weights during training, and may require many exposures to a training set before a network performs appropriately. Even if a network does learn a task, it may learn to generalize improperly, and not perform well on novel input patterns. Finally, if a network learns to perform as desired on a given problem, there is not generally a way to communicate in abstract terms *how* the network has solved the problem. That is, connectionist networks cannot provide a simple explanation of their solutions. However, in spite of their problems, connectionist networks have shown their usefulness on many tasks, especially low-level recognition and categorization problems, such as optical character recognition.

## 3.2 Connectionist networks

A *connectionist network*, sometimes called a *parallel distributed processing* (PDP) network, is a model which is based loosely on neural architecture. Connectionist networks attempt to capture the essence of neural computation: many small, independent units calculating very simple functions in parallel. These networks are composed of two basic building blocks: idealized neurons (often called *units*) linked via weighted connections. Each unit has an associated *activation* value, which can be passed to other units via the links with the connection weights mediating the amount of activation that is passed between units.

In a typical connectionist network, activation flows from unit to unit over the weights like electricity over wires. The units in *feed-forward* networks are organized into layers with no connections from one layer back to previous layers, so all activation flows in one direction without cycles. In most such networks, there are three types of layers: *input*, *hidden*, and *output*. If the units in a layer receive activation from outside the network, then the layer is called an *input layer*. An *output layer* produces activation

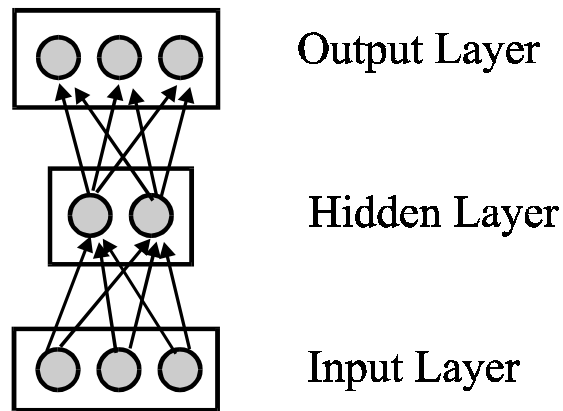


Figure 3-1. A 3-layer, feed-forward network.

representing the result of the network's computation, and a *hidden layer* is defined as a layer that is neither an input nor an output layer. Furthermore, layers can be subdivided into sections called *banks*.

Figure 3-1 shows a schematic diagram of a standard 3-layer, feed-forward network.<sup>4</sup> Layers are usually displayed with input layers on the bottom and output layers on the top, as shown. The gray circles represent units, the rectangles represent layers, and the arrows represent weighted connections. Usually when networks are large, not all of the weighted connections are drawn.

Briefly, processing in a network occurs as follows. The input layer is loaded with a set of activations called an *input pattern*. Activation flows across the weights between the input and the hidden layers. At each unit in the hidden layer, the incoming activations

---

<sup>4</sup> The number of layers in a network is sometimes calculated by not counting the input layer. This is often warranted as the input layer does not do any computing but serves only as a place to load the input values. However, I will count input layers whenever I describe a network architecture. Therefore Figure 3-1 shows a 3-layer network.

are summed and passed through an activation function. The hidden activations then flow from the hidden layer to the output layer. At each unit in the output layer, the new activation is calculated as before. The resulting activations emerge from the network as the *output pattern*. The process of propagating activation from the input layer through the hidden layers to the output layer is called the *propagation phase*.

Figure 3-2 shows a single unit  $m$  of a connectionist network. The total activation coming into a unit is called the unit's *net input*.<sup>5</sup> The net input of unit  $m$  is calculated by summing the incoming activations multiplied by the associated weights, and is labeled

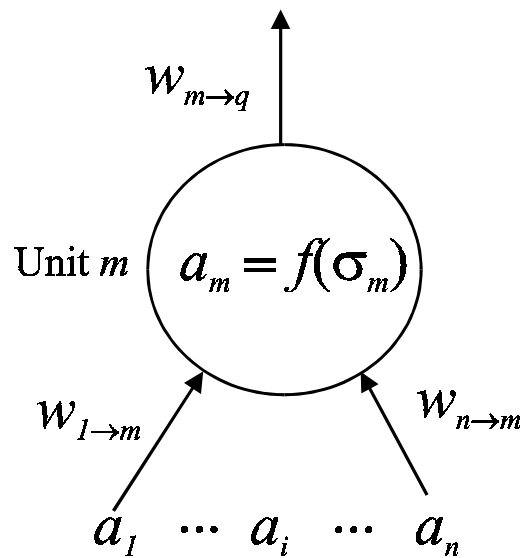


Figure 3-2. An idealized neuron. Activations come in from units 1 through  $n$ , and are propagated to unit  $q$ .

<sup>5</sup> Here, ‘net’ is not short for ‘network’ but refers to the “total left after deductions,” as in *net income*.

$\sigma_m$ . The net input is only computed for hidden and output layers, not input layers. For input layers, the activation is defined to come from an external *input pattern*. The net input to a unit  $m$  in the hidden or output layer is thus defined to be:

$$\sigma_m = \sum_{i=1}^n a_i w_{i \rightarrow m} \quad (3-1)$$

where  $i$  is the index of the units in the previous layer that are connected to unit  $m$ ,  $a_i$  represents the activation of unit  $i$ , and  $w_{i \rightarrow m}$  represents the weight of the connection from unit  $i$  to unit  $m$ . The unit's *activation function*,  $f$ , is then applied to the net input,  $\sigma_m$ , yielding the new activation value. Typically,  $f$  is a logistic function of the form:

$$f(\sigma_m) = \frac{1}{1 + e^{-\sigma_m}} \quad (3-2)$$

Applying the logistic function to the net input  $\sigma$  squashes it down into the range from 0 to 1. Figure 3-3 shows the squashing function with  $\sigma$  plotted on the x-axis, and  $f(\sigma)$  on the y-axis. The squashed net input becomes the new activation for unit  $m$ , and is used to calculate other net inputs in subsequent layers (e.g., in Figure 3-2, the activation is passed on to unit  $q$ ). This particular squashing function is used because it is non-linear and its first derivative has special properties that are used in the learning update procedure described below.

### 3.2.1 Linear networks

As shown in Figure 3-1, between each pair of layers is a set of weights. Each weight matrix transforms the activations from the lower layer to the activations at the next layer. In a *linear network* (a network where  $f$  is a linear function) there is no need for a hidden layer because any function that can be computed with two linear transformations can, of course, be computed by a single linear transformation (i.e., a single weight matrix between two layers). Such two-layer networks are limited in the functions they can compute (Minsky and Papert, 1969).

### 3.2.2 Non-linear, 3-layer networks

Although two-layer networks are restricted in their computing power, non-linear, 3-layer networks are potentially much more powerful. This type of connectionist network is capable of developing patterns of activations on the hidden units in response to a learning algorithm. These hidden layer patterns can be viewed as *recodings* or *representations* of the network's input patterns. These representations, combined with a differentiable, non-linear activation function, allow a network to solve problems which

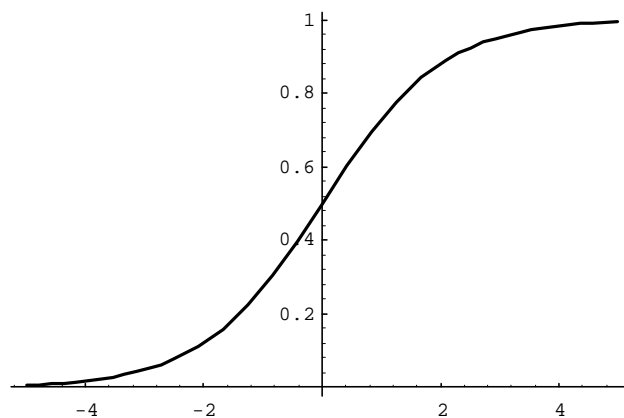


Figure 3-3. A simple logistic function. The x-axis shows the inputs into the function, and the y-axis the resulting squashed value between 0 and 1.

are out of reach of linear networks. The next section examines how weighted connections can be altered in a network so that it may learn.

Table 3-1. A sample set of input/output training pairs

	Input Pattern	Target Output Pattern
1	0 0 0	0 0 1
2	0 0 1	0 1 0
3	0 1 0	0 1 1
4	0 1 1	1 0 0
5	1 0 0	1 0 1
6	1 0 1	1 1 0
7	1 1 0	1 1 1

### 3.3 Learning via Back-propagation

Before training, a connectionist network, such as the one described in the previous section, usually doesn't perform any useful function. The weights are generally initialized to random values, so the network produces meaningless output when activations are propagated from the input layer to the output layer. However, by using an iterative learning algorithm to adjust the weights, the network can be gradually coaxed into producing a desired output pattern for each input pattern in its training set.

There are many algorithms for adjusting weights, such as competitive learning (von der Malsburg, 1973; Fukushima, 1975; Grossberg, 1976), and genetic algorithms (Holland, 1975). Possibly the most widely used method is the *back-propagation of error* method, or simply 'backprop' (Rumelhart, Hinton, and Williams, 1986). Backprop is a gradient descent learning algorithm. In a nutshell, back-propagation calculates an error value for each output unit and adjusts the weights responsible for the error in such a way

as to reduce it the next time around. The following is a more detailed sketch of the back-propagation algorithm and its use.

When training a network with backprop, each output unit has a desired output value called the *target*.<sup>6</sup> The difference between the target value and the actual output value produced by the propagation phase is called the *error*. The main function of the backprop algorithm is to correctly assign blame to the weights responsible for the error and to adjust them accordingly.

As an example, consider the patterns shown in Table 3-1. These patterns define the function ADD1 for 3-digit binary numbers. A network that correctly transforms the input patterns to their associated output patterns could be described as computing the binary ADD1 function. For instance, input pattern #1, the binary representation of zero, should produce the target output pattern {0 0 1}, the binary representation of one. Using the network shown in Figure 3-1, the input/output patterns shown in Table 3-1, and the backprop learning algorithm just outlined, the network can be trained to produce the desired target output patterns, thereby implementing the ADD1 function. A specific examination of this process follows.

Initially, all 12 of the weights in Figure 3-1 are set to small random values.<sup>7</sup> The back-propagation training procedure begins with the random selection of a pair of patterns, say pair #2. The first step is to load the input pattern onto the input layer. This is

---

<sup>6</sup> Because analogies are divided into source and target components, I will refrain from calling the desired outputs of a network “targets” so as to avoid confusion. In subsequent chapters I will refer to them as simple “desired outputs”.

<sup>7</sup> Specifically, the weights should be small enough so that the net activation at any unit should initially be between 0 and 1.

---

accomplished by setting each input unit's activation to the corresponding value in the input pattern. For pair #2, the activations of the input units would be set to {0 0 1}. Next, the activation is propagated through the network as previously described. At this point, there now exists a pattern of activation on the output layer, say {0.4 0.6 0.7}. As the desired target output values are {0 1 0} the associated error values are {-0.4 0.4 -0.7}, i.e., the difference between the target and the actual output. Those error values are then "back-propagated" through the network, and the weights are updated accordingly.<sup>8</sup> For a detailed explanation of the workings of back-propagation see (Rumelhart, Hinton, and Williams, 1986) or (Bechtel & Abrahamsen, 1991).

The propagation of error and adjustment of weights for a single training pattern is called a *trial*. Following the first trial, another pair of patterns is selected, and the entire process is repeated. One such sweep through all training patterns is called an *epoch*. At the end of an epoch, the total error of all patterns is computed. If the total error is within an acceptable tolerance value, training stops, otherwise another epoch begins.

When the error falls within range of the stopping criterion, the network has successfully learned the training patterns. However, there are many possible ways in which the network may have learned the training patterns. The network might somehow have developed specific 'rules' for transforming each individual input pattern into its corresponding output pattern. If this were true, then the network would be unable to generalize. That is, the network would have found a way to produce the correct answer for each training pattern, but the solution would not carry over to new patterns it has not seen before. The generalizing ability of the network can be tested by placing an untrained

---

<sup>8</sup> To ensure convergence of the network to a stable, final set of weights, the updating of the weights will often be delayed until after all input/output pairs have been seen. This is called *batch-mode* training.

input pattern on the input layer, propagating the activation through the network, and comparing the actual output with the expected output.

For instance, suppose that we had removed pair #4 from the training corpus prior to training. After training, we place the input pattern {0 1 1} (pattern #4) onto the input layer. If the output layer's activations are sufficiently close to {1 0 0} (the target output pattern), then the network can be said to have generalized, that is, to have recognized and taken advantage of regularities in the input patterns it has learned. Many variables may effect whether a network generalizes, including the number of training patterns, the number of units in the hidden layer, the input representations, and difficulty of the problem.

Sometimes, statistical regularities in the input patterns appear in a way that is difficult (or impossible) for a network to take advantage of. However, back-propagation networks with a hidden layer have the capacity to *recode* their inputs. By doing so, they are able to reformulate the problem and sidestep many of the limitations initially pointed out by Minsky and Papert (1969). Many types of problems can be solved in this manner, given that one knows (or can compute) a target value for each output unit. For example, 3-layer back-propagation networks have learned to convert text to phonemes (Sejnowski and Rosenberg, 1987), learned to form the past-tense of regular and irregular verbs (Rumelhart and McClelland, 1986), and learned to discover regularities hidden within relationships (Hinton, 1988), to name just a few.

### 3.4 Connectionist Representations

In order to examine connectionist representations, we need the foundation of some basic ideas and a set of common terms. One of the most basic concepts in connectionism is that of *pattern*. A pattern is an ordered set of activation values. In this discussion, we

will use the word ‘pattern’ to mean a *static* set of activations, but a pattern could also be dynamic, existing through time.

A *representation* is a pattern that carries or conveys *meaning*. Meaning is a function that maps *entities* to representations. *Entities* are composed of *objects*, *attributes*, or *relations*. Objects, attributes, and relations are concepts used by the human designers to describe and perceive the world. A *representation scheme* is the inverse function of meaning; a representation scheme maps representations to entities. A *situation* is a set of entities. Although these terms may be defined circularly, together they form a consistent view of representations.

Let us now reexamine the ADD1 network using this terminology. The numbers 0 through 7 are the entities being represented. The number 2 is an entity represented by the pattern {0 1 0}, and the representation {0 1 0} means the number 2. As the patterns are constrained to consist of the activation values 0 and 1, the patterns are called *binary*.

Table 3-2. A localist representation scheme.

Pattern #	Input Pattern	Target Output Pattern
1	1 0 0 0 0 0 0	0 0 1
2	0 0 0 0 0 1 0	0 1 0
3	0 1 0 0 0 0 0	0 1 1
4	0 0 0 1 0 0 0	1 0 0
5	0 0 1 0 0 0 0	1 0 1
6	0 0 0 0 1 0 0	1 1 0
7	0 0 0 0 0 0 1	1 1 1

There are two schemes for encoding entities: *localist representation schemes*, and *distributed representation schemes*. A localist representation scheme is a function that maps all of the relevant entities from the world to a representation such that all of the representations of the relevant entities are orthogonal. As an example, consider a new representation scheme for the ADD1 network. Notice that the original representation scheme used to encode the numbers 0 through 6 produced non-orthogonal patterns (Table 3-1) and is, therefore, not a localist representation scheme. One possible localist representation scheme for the ADD1 network is shown in Table 3-2. A distributed representation scheme is any function that produces non-orthogonal representations for all relevant entities. There are many ways for a set of representations to be non-orthogonal; Table 3-1 is one set, and Table 3-3 is yet another quite different distributed representation scheme. Analogator will use only localist representations for input patterns.

Table 3-3. Another distributed representation scheme.

	Input Pattern	Target Output Pattern
1	0.1 0.5	0 0 1
2	0.4 0.3	0 1 0
3	0.3 0.7	0 1 1
4	0.6 0.9	1 0 0
5	0.7 0.8	1 0 1
6	0.5 0.4	1 1 0
7	0.2 0.1	1 1 1

### 3.5 Hidden Layer Patterns

As mentioned, a 3-layer connectionist network trained with backprop has the capability of non-linearly recoding each of the input patterns into non-arbitrary, distributed patterns of activation at each successive hidden layer. This process of learning

to recode input patterns into intermediate patterns of activation spread across the hidden layers amounts to the development of distributed internal representations of the input information by the network itself. The ability of connectionist models to develop their own distributed internal representations is an extremely important property of this class of models.

### 3.6 Principal Component Analysis

Although much power is gained by having non-linear hidden layers, the networks lose perspicuity; analysis of how a network solves a particular problem can be very difficult. To get a better idea of how a network solves a particular problem, one can examine its hidden layer patterns as follows.

In a 3-layer network, each input representation produces a pattern on the hidden layer units on its way to the output layer. If there were a small number of units on the hidden layer (e.g., less than four), one could examine these activations by assigning each unit an axis in a graph. For each input representation, the hidden layer pattern would define a point in *hidden layer activation space*. Plotting each input pattern's position in hidden layer activation space can provide visualization hints as to how the network has solved a problem. For instance, one might see clusters of points. These clusters presumably reflect abstractions and categorizations discovered by the network and represented in the hidden layer patterns.

Plotting patterns in hidden layer activation space is only useful, however, when the number of the units in the hidden layer is three or less as humans are not very good at analyzing data in four or more dimensions. However, Principal Component Analysis (PCA) can provide an approximation of a multi-dimensional space in fewer dimensions. PCA works by finding dimensions that better describe the variance among a set of points in a multi-dimensional space than the original axes. For instance, consider the points on the left-hand side of Figure 1-2. When PCA is applied to the set of points, a vector is found that maximizes the differences between all points (the dashed arrow). This vector,

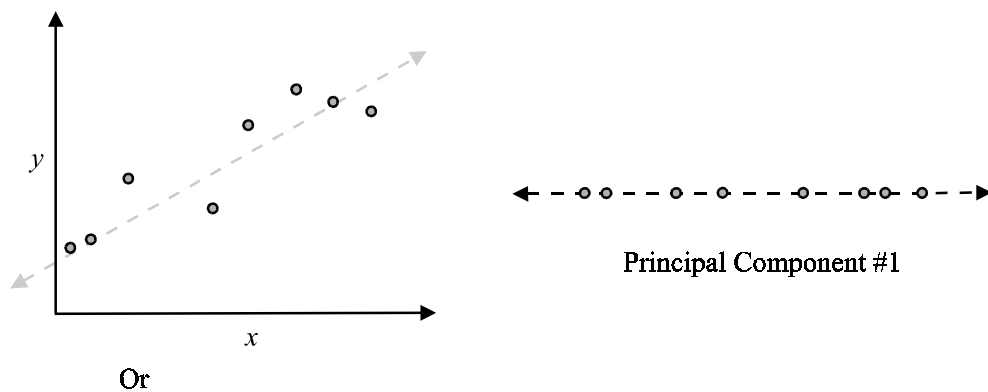


Figure 3-4. Principal Component Analysis. The dotted line accounts for almost all of the variability between the points.

called an *eigen-vector*, defines the first principal component. On the right-hand side of Figure 3-4, those same points are presented along a single dimension, the first principal component. Thus, the original  $x$ ,  $y$  plot has been reduced to a single dimension that captures most of the variance of the original points. The PCA algorithm is repeated to find the dimension with the next most variance, and so on. Of course, the real usefulness of PCA comes from reducing a highly dimensional representation down to two or three principal components, as demonstrated in Chapter 5.

### 3.7 Simple Recurrent Networks

Feed-forward networks work well for associating a single pattern to another. However, allowing cycles in the flow of activation in a network gives the network the ability to learn patterns in *sequences*. Networks that can learn patterns in sequences are useful for handling data in time, or variable length patterns. Analogator uses sequencing to allow hardware sharing between analogous components, as we will see in the next chapter.

To train a recurrent network to the true gradient using backprop, information must be kept about the flow of activation over a recurrent connection through time. Elman has proposed a short cut to the process and it has been found to work quite well (Elman, 1990). As this solution simplifies the calculations by approximating the true gradient descent, Elman calls this type of network a Simple Recurrent Network, or SRN.

SRNs eliminate the need to keep detailed histories about recurrent connections by adapting the standard feed-forward architecture to accommodate the recurrent flow of activation. Elman's method is described as follows. After the propagation and back-

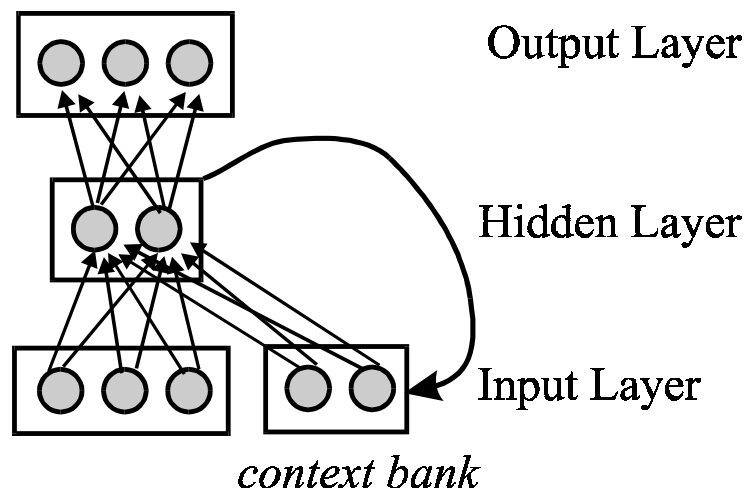


Figure 3-5. A Simple Recurrent Network (SRN).

propagation phases, the activation patterns from the hidden layer are copied to a special bank of units on the input layer, called the *context bank*. At the next time step, input will flow from the regular input units and the context bank to be combined at the hidden layer units. Following the propagation phase and the backprop phase, the activations will again be copied back to the input layer, and so on after each step.

This method has no guarantee of converging as it only approximates the true gradient descent, but it has been found to work quite well in practice. This has allowed researchers to work with feedback networks without the trouble of the more complex, and more time consuming, learning algorithms. SRNs have been applied to grammar learning (Elman, 1990), robot control (Meeden, 1994), and many other problems requiring recurrent processing. SRNs are related to more elaborate connectionist networks, such as Pollack's Recursive Auto-Associative Memory (RAAM; Pollack, 1988) (see also Blank, Meeden, and Marshall, 1992, for a thorough examination of sequential RAAMs, a close cousin of the SRN).

### 3.8 Tensor products

Many methods have been developed in the past few years for representing structured information in a connectionist network. This has, in part, been done to meet the challenges posed by Fodor and Pylyshyn (1988). They charged that connectionist models were incapable of adequately representing structure. One early answer to their challenge was Smolensky's *tensor product variable binding* (Smolensky, 1991). Let us briefly examine the problem of representing structure in connectionist networks.

We have seen how a pattern may represent an entity, but what must one do to represent two entities? One answer would be to simply assign a new pattern to represent the pair. But this quickly creates an explosion of patterns as we would need a unique one

for each combination of entities. Another answer might be to concatenate the two patterns, one after another. However, this creates a pattern that is twice as big as the others and would grow longer with each additional entity. What is desired is a fixed-width representation capable of representing one or more entities and the relationships between them. Also, we would like a system with a graceful degradation in performance as the number of entities increases. Although there now exist many such techniques for representing structured representations, the simplest is probably Smolensky's tensor product method.

Smolensky's method first requires that structures be broken down into *roles* and *fillers*. A role is a named position; a filler is the specific object or thing that fulfills a particular role. For example, 'president' is a role currently filled by the filler 'Bill Clinton'. Roles and fillers are a very common form of representation used by linguists and computer scientists.<sup>9</sup> The tensor product method is a mathematical procedure for associating (also called *binding*) a role with its filler.

Specifically, Smolensky's tensor product representation is a distributed set of activations formed by taking the outer product of a role pattern with a filler pattern (Smolensky, 1990) (see Figure 3-6). In this methodology, a role is an entity, as is the role's filler (i.e., they both have patterns). The matrix of values formed by calculating their outer product represents the *binding* of these two entities. For instance, in Figure 3-6 the role vector is represented by the activation values {0.9 0.1 0.5 0.1} and the filler is represented by {0.1 0.5 0.9 0.1}. The matrix produced by the outer product represents the filler in that particular role.

---

<sup>9</sup> Computer scientists also call roles *variables*, and their fillers *values*.

Multiple role/filler pairs may be represented by adding outer product-formed matrices. In this manner, one may represent complex structures, such as trees or lists. For instance, consider the sentence “Mary gave the book to John.” One possible set of role/filler pairs is: *GIVER* / *mary*, *RECEIVER* / *john*, and *OBJECT-OF-TRANSFER* / *book*, where roles are given labels in uppercase letters, and their fillers are shown in lowercase italics. Each filler is bound to its role via an outer product of the two associated patterns. Finally, all three outer products are summed to produce a single matrix representing the entire sentence or structure (see Figure 3-7). The 3 by 3 matrix to the far right of Figure 3-7 represents the full sentence, “Mary gave the book to John.” The number of vectors used in creating a tensor determines its dimension, which is also called its *rank*. Therefore, Figure 3-7 shows the creation of a rank-2 tensor.

Notice that the tensor product method creates representations that are larger in rank than their original vectors (i.e., the resulting matrices are substantially larger than their originating role and filler patterns). This problem has been addressed by Plate’s Holographic Reduced Representations (HRR) and Kanerva’s spattercode (Plate, 1991; Kanerva, 1996). However, their basic methodology is not substantially different from

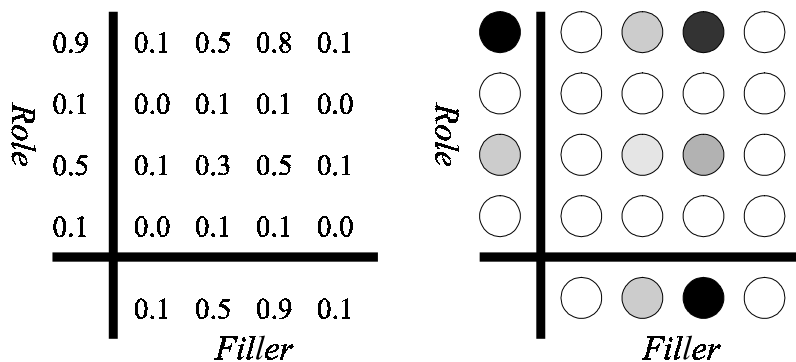


Figure 3-6. Two methods of depicting a tensor product role/filler binding.

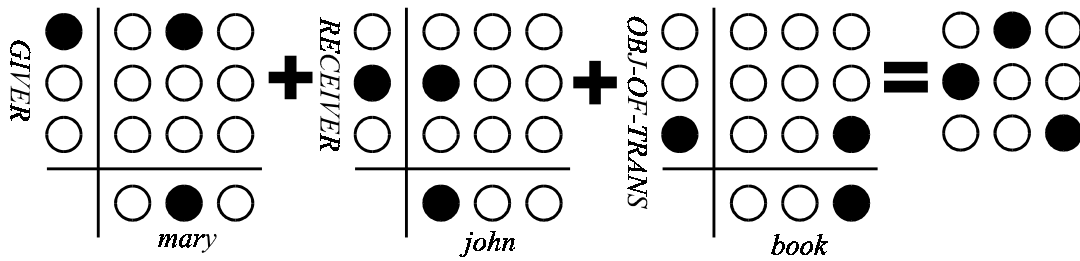


Figure 3-7. Tensor product representation of "Mary gave the book to John."

Smolensky's tensor product variable binding. We will further compare these methods in Chapter 6.

In this chapter we have examined the connectionist foundations of Analogator, namely: the basic architecture of connectionist networks, a learning algorithm called back-propagation, the development of hidden layer patterns and their analysis, a network architecture for simple recurrent networks, and role-filler binding via tensor product algebra.

We now turn to see how these basic building blocks are used to create Analogator.