University of Rochester Working Papers in the Language Sciences—Vol. Spring 2000, no. 1
Katherine M. Crosswhite and Joyce McDonough (eds.)

Connectionist Modeling for…er… linguists

Bob McMurray (`mcmurray@bcs.rochester.edu`)
Department of Brain and Cognitive Sciences
University of Rochester, Meliora Hall, Rochester, NY 14627 USA

Abstract

Connectionist modeling (AKA **neural network modeling, connectionism**) is rapidly becoming a dominant descriptive and theoretical tool for the psycholinguist. Below is a brief introduction to some of the terms and concepts used in connectionist modeling. Connectionist models are no different than any other sorts of theories in cognitive science, they merely offer a new computational toolbox, or set of algorithmic constraints on models and theories of cognitive phenomena. In this paper I review many of the important components of connectionist models and introduce some of strengths, pitfalls and caveats that casual readers and serious modelers must be aware of.

Introduction

If you've read past the abstract, you must have resisted the urge some linguists feel to put down the article after reading the word "connectionist". Thank you. I'd like to welcome you to our informal field by teaching you some of the lingo you'll need to navigate. I'll try to avoid using the math that modelers love to flaunt and instead focus on the underlying concepts and architectures. Hopefully, after reading this, you will be able to start reading modeling papers and understanding much of what is going on. Moreover you may stop falling asleep at modeling talks. If I'm lucky, you may even collaborate with psycholinguist to build your own models of linguistic phenomena. Hopefully you'll have enough understanding of the basic terms and issues to do all of these things after reading this paper.

Throughout this paper I've tried to put most connectionist terms in **boldface** so that you can find particular concepts quickly by scanning. **Connectionism** as a field grew out of work in neurobiology, computer science, electrical engineering, statistics, and cognitive psychology (and probably other fields), so there are often many terms that mean the same thing (depending on what your background is). In these cases, I have tried to provide all of the terms. I've also tried to include terms and concepts that are not formally defined anywhere, but have proven useful to connectionists discussing their work over the years.

A common question that linguists have asked me is "what *is* a **connectionist model**?" The answer to that question is surprisingly quite simple. A **connectionist model** is really an algorithm for turning some **input** (which presumably maps onto something of psychological or linguistic interest) into some other **output** (which may map onto some data). In this regard it is very similar to any other cognitive or linguistic model that has been implemented computationally. Take, for example, an Optimality Theory Grammar. An OT grammar turns a collection of phonological forms from Gen (the input) into the actual production (the output). The only difference between this grammar and a **neural network** is that the kinds of **computations** we are allowed to use in creating the algorithm are different. OT prescribes one type of computation (constraint satisfaction), while **connectionist models** use computations that are very loosely based on the kinds of computations that neurons and populations of neurons might perform. Under this view, **connectionism** is simply a set of (mostly) agreed upon guidelines for what sorts of algorithms are appropriate for describing cognitive behavior.

Architecture

All connectionist models are composed of two simple concepts: **nodes** (AKA **neurons** or **units** or **cells**) and **weights** (AKA **connections** or **synapses**).

A **node** can be considered a *highly* idealized representation of a neuron. It has an **activation** (or **firing rate**) that tells us how strongly that neuron is firing. In a very simple case, a **node** might be assigned to a real world concept such as a specific phoneme, /b/. It's neighboring nodes may represent other phonemes, /d/ and /t/. In this case, the **activation** of the /b/ node relative to the other nodes would tell us how strongly the system believes a /b/ was present in the input. Oftentimes the **activation** of a **node** will be simplified by saying the **node** is either **on** (**firing**) or **off** (**not firing**, **inactive**). Keep in mind that very few **connectionist models** have **nodes** with discrete **activation levels—on** or **off** simply refer to the **node** having a lot of **activation** (relative to the other **nodes**) or a little.

Nodes are organized into **layers** (AKA **arrays** or **vectors**). Each **layer** is a cluster of **nodes** that are [usually] functionally related. For example, one layer of a network may consist of the group nodes that correspond to each phoneme; another layer may have nodes that correspond to words.

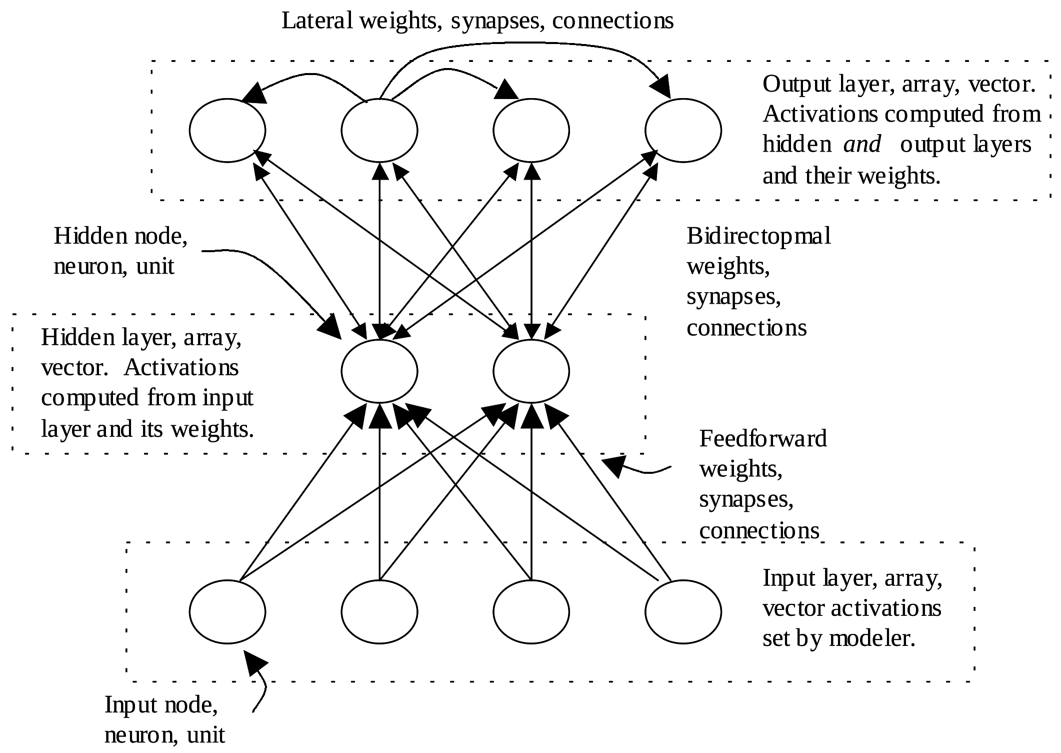
Lateral weights, synapses, connections
Output layer, array, vector. Activations computed from hidden *and* output layers and their weights.
Hidden node, neuron, unit
Bidirectopmal weights, synapses, connections
Hidden layer, array, vector. Activations computed from input layer and its weights.
Feedforward weights, synapses, connections
Input layer, array, vector activations set by modeler.
Input node, neuron, unit

Figure 1: A typical diagram of a neural network with its features labeled.

In any model, one or more **layers** is designated the **input layer. Stimulus** from the outside world is received into the network via this **input layer**. The **stimuli** consist of numerical representations of real world objects or stimuli. When the modeler sets the **activations** of the **nodes** of the **input layer** to match one of these representations, the network has received that stimulus as its input. The patterns of **input activation** may come from a corpus of text, a digitized waveform, or any other set of stimuli the modeler wishes. Additionally, they could be manually set to arbitrary values if the modeler wishes to abstract away from real input (possibly the real input is too complex to illustrate the problem the modeler wishes to work with). The set of **input activation levels** the modeler decides to use is called the **training set**. Each item in a **training set** minimally consists of the activations for each **input node** in the **input layer**. The **training set** will sometimes contain other information such as the expected value of the output nodes for each input. This will be discussed when we talk about learning.

Each network will also have one or more **output layers.** The **output layer** is the cluster of nodes that will determine the network's "behavior". The values of these nodes are the values that we will attempt to relate to the empirical data that we are trying to evaluate. For example in a network designed to categorize phonemes, the **input layer** might represent a digitized waveform, and the **output layer** would have a node corresponding to each phoneme. The way in which the **activation** of **nodes** in the **output layer** is related to the empirical data or behavior is called the **linking hypothesis** (because it links models and data). For example, for our phoneme categorization example, our **linking hypothesis** might be that the model will choose the phoneme with the most **activation** as the phoneme it heard. I'll talk more about **linking hypotheses** later.

Layers of nodes that do not receive input or provide output are called **hidden layers.** These layers compute some sort of intermediate representation (between **input** and **output layers**). Many modelers dispense with the **input, hidden,** and **output layer** designations all together and simply refer to layers by what they designate. The TRACE model (McClelland and Elman, 1986), for example has a **feature layer**, a **phoneme layer,** and a **word layer,** but none of them is designated the **output layer.** TRACE, in fact, can use either phonemes or words as the output depending on the task at hand. In models like these, one must think about the logical flow of information is a psychological sense if you wish to determine the **input** and **output** layers. Many models are described simply as **2-layer** or **3-layer** networks (or more). A **2-layer** network will necessarily have only an **input** and **output layer.** A **3-layer** network will have both of these plus one **hidden layer.** A **4-layer** network will have two **hidden layers.**

In the remainder of this paper, whenever I refer to simply **input** or **output,** I will be referring to the entire **input** or **output layers** (i.e. the **pattern** of **activations** of across node in the **layer**).

Often times, a **layer** of nodes is thought of as a set of coordinates in a **multidimensional space**. This is easiest to visualize for a network of two nodes. The activation of the first node could be considered the X-coordinate. The activation of the second node would be the Y-coordinate. Then any particular **pattern of activations** across the two nodes can be thought of as a unique point in a 2-D coordinate system. So if the input activations for the two nodes were .5 and .8, we could talk about the input as the single point <.5, .8>.

Of course, when we move up to larger networks we won't be able to visualize a 16 dimensional space. However, we can still talk about one, and this spatial metaphor is used frequently. Under this metaphor, the **input space** would consist of all regions of the possible N-dimensional space that are used in the network (where N=number of inputs). The **output-space** is the corresponding regions in

M-dimensional space (where M=number of output nodes). People often refer to the **dimensionality** of a space (which is simply the number of nodes). Then when information is passed from an **input space** of **high dimensionality** to an **output space** of **lower dimensionality,** the information is undergoing **dimensionality reduction—**it must be compressed (and some information invariably lost) in order to "fit" in the lower dimensionality space. This forces the network to make **group** some **inputs** together and discard others according to the correlations it finds in its inputs. They types of categorizations it makes may be of ultimate interest psychologically.

This way of describing network behavior spatially provides a convenient way of describing a network. When activation patterns change, we can talk about the network moving to a new point in the **input space.** Moreover modelers often speak of **learning** (which I will discuss shortly) as a search through the **output space.** Finally, **dimensionality reduction** is often thought of as a form of information compression (as a network may have to represent 3-D information, for example, in only two dimensions). **Dimensionality reduction** is also a common concept used to describe statistical techniques such as factor analysis, clustering, and multidimensional scaling (if you don't know these terms, that's fine, I merely throw them out to show that the analogy can be helpful in relating **neural network** computations to other types of computational tools).

In a network **nodes** are connected to each other by **weights** (AKA **synapses**, **connections**). Each **weight** represents the amount of **activation** that can be passed by one **node** to another. If an **input node** is highly active and it has a strong **connection** to an **output node**, that **output node** will also be highly **active**. If it has a weak **connection** that **output node** will not be highly **active.** We'll go over the details of this in a moment.

The set of all weights between two layers is termed the **weight matrix** (for reasons we'll see shortly). When a model is built, the **weight matrix** often starts as a matrix of small random numbers (as we will discuss, it will be modified later by **learning**).

Weights can either **excite** (make active) or **inhibit** (make inactive) the nodes they connect. **Excitatory weights** will cause a node to become more active if the nodes that connect to it are active. **Inhibitory weights** will cause a node to become less active if the nodes that connect to it are active.

Weights that pass information from **input** to **output nodes** (or in that direction between **hidden nodes**) are considered **feed-forward connections**. Weights that pass information backwards from **output nodes** to **input nodes** (or in that direction between **hidden nodes**) are considered **feedback connections. Bidirectional weights** pass information both ways. **Weights** that connect units *within* a **layer** are considered **lateral connections.** The most common use of **lateral connections** is **lateral inhibition** in which nodes within a layer attempt to turn each other off. The result of this process is that a few nodes have all the activation and the others have none.

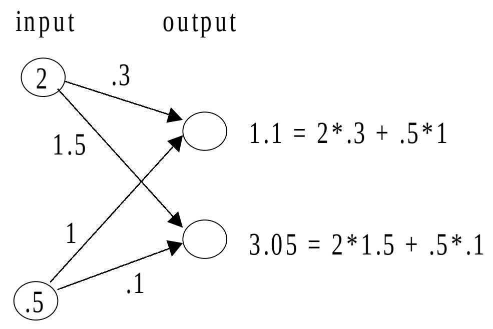

Figure 2: A basic connectionist network. Circles represent nodes, and arrows weights. Values inside circles represent activations.

Consider the example network in figure 2. This network consists of two **input nodes** and two **output nodes** (a 2x2 network), **fully connected** (each input nodes is connected to each output node) and **feed-forward**. The **activations** of the input nodes have been set to 2 and .5 by the modeler.

To compute the values of the **output nodes**, we will use some function of the inputs and the weights. This function is called the **activation function.**

$$\text{output}_{\text{top}} = f(\ \text{input}_{\text{top}}, \text{input}_{\text{bottom}}, \text{weight}_{\text{top->top}}, \text{weight}_{\text{bottom->top}}) \qquad (1)$$

The simplest **activation function** is the **linear activation function.** Each output node is simply the sum of the activation each input node multiplied by the corresponding connection (weight) to that output node.

$$\text{output}_{\text{top}} = \text{input}_{\text{top}} * \text{weight}_{\text{top->top}} + \text{input}_{\text{bottom}} * \text{weight}_{\text{bottom->top}} \qquad (2)$$
$$\text{output}_{\text{bottom}} = \text{input}_{\text{top}} * \text{weight}_{\text{top->bottom}} + \text{input}_{\text{bottom}} * \text{weight}_{\text{bottom->bottom}}$$

This can be generalized to:

$$\text{output}_y = \sum_{x=1}^{\text{Num input}} \text{input}_x * \text{weight}_{x\text{->}y} \qquad (3)$$

We can simplify this even further with some linear algebra. Let Output (with no index) become a **vector** of all the **output activations**, and Input (with no index) be a **vector** of all the **input activations.**

$$\begin{aligned} \text{Input} &= [\text{Input}_{top} \quad \text{Input}_{bottom}] = [2 \quad .5] \\ \text{Output} &= [\text{Output}_{top}\ \text{Output}_{bottom}] \end{aligned} \tag{4}$$

Now let **W** be defined as a **matrix** where the row indicates the **index** of the input node (in this case, the top node would have an index or row of 1 and the bottom would have an index of two), and the column indicates the **index** of the output node. The value at each position indicates the connection strength or weight.

$$W = \begin{bmatrix} \text{weight}_{1,1} & \text{weight}_{1,2} \\ \text{weight}_{2,1} & \text{weight}_{b2,2} \end{bmatrix} \tag{5}$$

$$W = \begin{bmatrix} \text{weight}_{top\text{->}top} & \text{weight}_{top\text{->}bottom} \\ \text{weight}_{bottom\text{->}top} & \text{weight}_{bottom\text{->}bottom} \end{bmatrix}$$

$$W = \begin{bmatrix} .3 & 1.5 \\ 1 & .1 \end{bmatrix}$$

Then by the definition of matrix multiplication (which essentially says: for each output node, do equation 3, and concatenate all the results into a vector) we can simplify the whole thing into.

$$\text{Output} = \text{Input} * W \tag{6}$$

where '*' indicates matrix multiplication, and Output and Input are vectors, W is a matrix. As some one to explain the linear algebra to you, and you will see it's not too complicated. You should recognize, thought, that equation 6 and equation 3 are doing the same thing, as you will often see it notated both ways.

All of this stuff so far has been to describe the **linear activation function.** This activation function says that as you give **input activation** to an output node (as a function of the **weights**) the **output activation** will increase proportionally. This isn't the only possible **activation function,** though. As equation 1 implies virtually any function could be used (although modelers tend to limit themselves to simple, understandable functions that may be neurologically plausible).

The most common **nonlinear activation function** (i.e. not equation 3) you will see is the **logistic activation function.** Without going into the math much, the

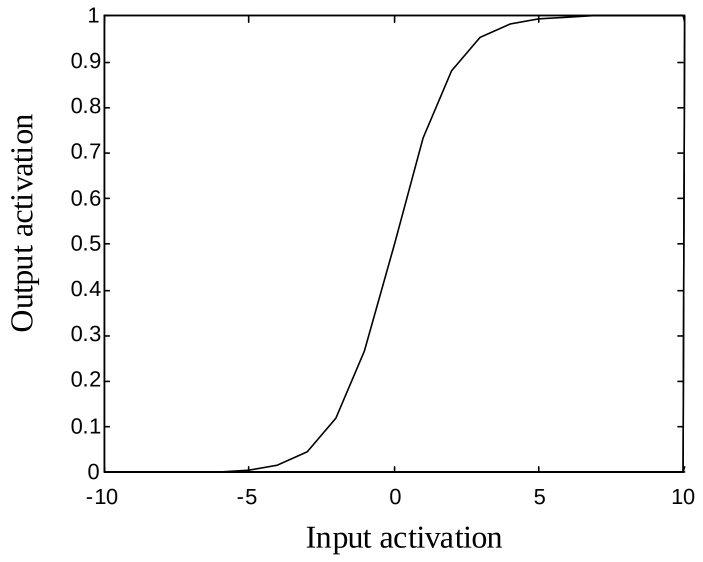

Figure 3: The logistic activation function. For any input activation to an output node, the logistic function outputs a value between 0 and 1.

logistic activation function serves to truncate the possible values of the output activation to a value between 0 and 1. If the sum of inputs*weights is high, the output node will equal 1. If that sum is low, the output node will have an activation of 0.

Non-linear activation functions are crucial to the success of **multiple-layer networks** because it has been shown that for any network with more than two layers that uses a **linear activation function**, a **two-layer** network can be built that performs equivalently. Essentially, if you want to reap any advantage out of having more than two layers, you have to use a **non-linear activation function.** The **logistic function** is a particularly good one, since the **logistic function** is what is known as a **basis function.** A **basis function** is a function that can approximate any other function if you add enough of them together (the **Gaussian curve**, and the **sine wave** are other examples of **basis functions**). So, if you think of a bunch of **hidden units** with **logistic activation functions** a network could approximate many other functions by simply adding them together. Because of this, neural networks have been termed **universal function approximators**. Although often **connectionist models** have been associated with **non-modular (or interactive)** theories of processing**,** and ***tabula-rasa,*** **statistically-oriented** theories of learning, as **universal function approximators**, **connectionist**

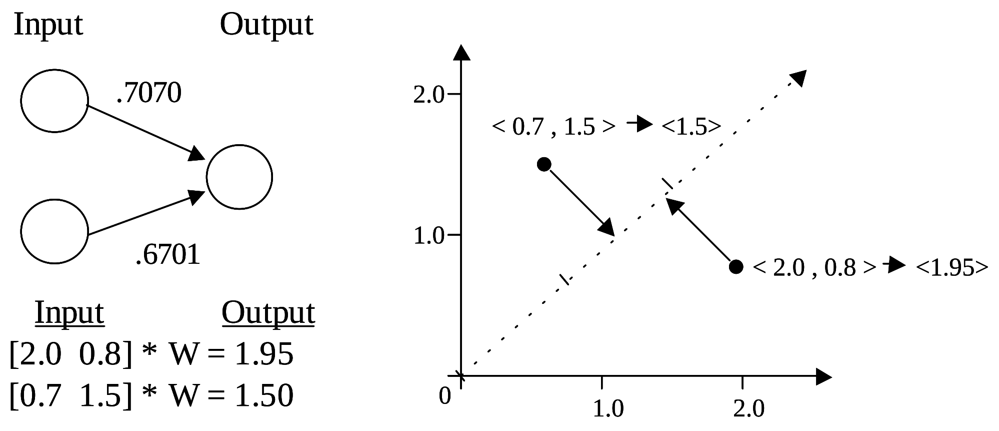

Figure 4: A neural network trained to perform a dimensionality reduction and the corresponding "geometric" representation of that reduction. In this case, the network is mapping the points in two dimensional space to a point in one dimensional space (along the dotted line). The values of the weights determine the equation of the line. Of course, not all networks perform a dimensionality reduction. Some keep the same dimensionality (merely shuffling the points in a predictable way), others will increase the dimensionality. The main point, though is that the weight matrix serves to perform this "remapping".

models can instantiate any sort of theory and should not be pigeon-holed into these particular lines of thought.

As I mentioned previously, **layers** of **nodes** are often thought of as **coordinates** in a **multidimensional space.** Under this view, the **weight matrix** then performs a **remapping** of a coordinate in N dimensional space to one in M dimensional space (where N is the number of input nodes, and M is the number of output nodes (see figure 4 for an example and explanation of this).

Representation

It is often useful to classify a model (or sometimes just a layer of a model) according to how it represents real world information.

A **localist representation** is one in which each node has a label of some kind, and when that node is active, it is in a sense saying "I think my label is correct." An example of this is a layer of cells in which each node corresponds to a different phoneme, or one in which nodes correspond to various people. Often localist nodes are derogatorily called **Grandmother Cells**, after a famous thought experiment in which someone asked "What would happen if your grandmother cell was damaged? Would you be unable to recognize your own grandmother?"

Although a trifle silly, this question does raise the important point that localist representations are very susceptible to damage. If your only cell that recognizes /b/ is damaged, your network can no longer recognize that sound any more. Many other people have pointed out, however, that a node doesn't necessarily stand for one neuron, but that it could stand for a whole population of neurons. Under this view, localist networks could easily survive damage.

A **distributed representation** represents information across several cells. Sometimes this is completely **arbitrary**. The word "boy" for example may be the pattern [1 1 0 1 0 0] (for a layer with five nodes), while "botch" is [1 1 1 0 1 1]. Other types of **distributed representations** may assign smaller units of meaning to individual nodes, although the interesting meanings are distributed across them. In our previous example, if the first cell responds to a "b", the second to an "o" the third to a "t", the fourth to a "y" and the fifth and sixth to a "c" and "h" respectively, our representation of each word is still **distributed**, but each unit is now meaningful as well. **Distributed representations** are particularly valuable in that they can withstand damage well (if you knock out a single node, there may be enough information remaining in the other nodes to maintain the representation). They also implicitly encode similarity. In the example above, "boy" and "botch" are similar in that they both share the first two letters. As such, their distributed representations share two active nodes. Because of this similarity encoding, **distributed representations** can often generalize patterns they have seen to novel ones.

Another type of **distributed representation** is the **topographic map** (AKA **population code).** In this scheme, a layer of cells represents the value of some continuous value by location. For example, in a layer of 10 cells that respond to sound frequency, the left-most cells may respond highest to low frequencies and the right most to high frequency. You can then recover which frequency the cells heard by looking at which cells fired. **Topographic maps** do not always represent their inputs linearly—they may have a lot of cells devoted to low frequencies and only a few to high frequencies, for example (this is what the output of a **Kohonen network,** which we'll read about shortly, looks like).

On some level, the debate over representation is a little pointless. It has been said that "one level of representation's localist representation is another level's distributed representation". The two representational schemes are not terribly different and really depend on the level of description you wish to use and the way in which you wish to describe your model's behavior and architecture. This is not say that it is not important to have a good understanding of the way in which your model represents information, just the there are no hard lines between distributed and localist representations, and one should not worry too much about the debate over them.

Learning

As you may have noticed, most of the interesting computational work in a neural network is done by the **weights**. At this point you are probably asking yourselves: how do I get the weights? That will be the topic of the next section: **learning.** I intend to keep to the more abstract conceptual level, however, an excellent description of the math behind the various learning systems can be found in Rumelhart, and McClelland (1986) and McClelland, and Rumelhart (1986). A good comparison of work in developmental psychology with connectionist learning can be found in Elman, Bates, Johnson, Karmiloff-Smith, Parisi and Plunkett (1992)

The **connection strength** associated with each **weight** is usually set by a learning process (although in some cases, such as TRACE, they can be set by hand by the modeler to implement a specific theory). Each network has a **learning rule** that essentially tells the network how to modify its weights at any given point. **Learning rules** change the **weights** as a function of the **activations** of the **input** and **output units**, the value of the **weight** itself and possibly some **error signal—** how close the actual **output values** are to the **target output values** (the ones you want the network to output). All **learning rules** have a component called the **learning rate** that determines how fast or slow the network can change its weights (essentially how much the network can change as a result of a single input). This gradual modification in **weights** leads to gradual change in the network's performance. The challenge to the modeler is to use learning rules appropriate to the task the model is given so that this change is an improvement. The process of modifying the weights over time is **learning** (also **training,** or simply **running** a model).

Regardless of the type of **learning rule** used, networks can be trained in two ways: **batch learning,** and **online learning.** In **batch learning**, the modeler presents the each item in the **training set** to the network and computes it's corresponding **output activations**. The **weights** are not changed until after the network has seen all of the possible **input/output pairs** when they will be modified using a **learning rule**. This forces the learning rule to consider all the input the network will ever see before changing any weights. The network will probably process the entire batch multiple times (each time is usually called an **epoch**, though this term is often misused in the literature). **Batch learning** is often considered implausible (e.g. it seems clear that children do not wait until they have heard every English sentence before learning to talk), but has the advantage of preventing a network from getting sidetracked by a single weird input.

The more common training scheme is **online** learning. In this scheme, the model cycles for multiple **iterations** (AKA **generations** and sometimes, confusingly, **epochs**). At each iteration, a single item from the **training set** is chosen (either randomly or by fiat), and the **activation** of each **input node** is set according to that item. **Output activation** is computed via the **weight matrix**, and the weights are modified via the **learning rule.** This is then repeated again and again until the modeler decides to stop. Usually the weights **settle** (stay at approximately the same value from iteration to iteration) after some time—this is a good place to stop training.

In most models, the model starts its "life" with a **random weight matrix** (essentially, each weight is a randomly selected value, usually within a small range). This ensures that the model does not start its life with any preknowledge of what it is to learn. It also is essential for many of the learning algorithms because initially, each **output node** will be biased differently in response to an input (if the network started out with a **weight matrix** consisting all the same number, each output node would be equally biased towards everything and learning would be very difficult).

So what kinds of things make up the learning rule? How does one know what to change the weights to? Modelers have been working on this issue for quite some time and have arrived at two broad categories of solutions: **supervised learning** and **unsupervised learning.**

Supervised learning rules change the weights as a function of a **teaching signal** which is provided by the modeler to tell the network what it *should* be outputting in it's output layer. This **teaching signal** is often considered part of the **training set.** For our dinky 2x2 network, the modeler might provide a training set such as the one below:

If the network sees…	…it should output
[1 0]	[1 0]
[0 1]	[1 0]
[0 0]	[0 1]
[1 1]	[0 1]

Then at each iteration, the **actual output** can be compared with the **target output** (the output provided by the **teaching signal**) and each weight can be adjusted according to whether it was contributing to the correct output or not. This comparison is usually in the form of an **error signal**, the difference between the **target** and **actual output.**

The **delta rule** (AKA the **LMS rule**) and **back-propagation** are two commonly used forms of supervised learning. The **delta rule** works very similarly to what I've described above. However, the **delta-rule** does not work very well for multiple-layer networks (unless you have **target values** for the activation of the **hidden units**). **Back propagation** is designed to send the **error signal** *back through* the hidden units (by transforming it via the weight matrix and a lot of messy calculus). Thus **back propagation** can be used with networks of any size. Since a complete description of this requires calculus, I will wave my hands a bit and move on to the next section. However, I direct the interested reader to Rumelhart, Hinton and Williams (1986). It is important to note, that **back propagation** is not widely considered to be **neurologically plausible** as a neurological mechanism for passing error information back through multiple **synapses** has not been found, and the, as I'll discuss later, the source of the **error signal** itself can lead to biological implausibility.

When doing **supervised learning,** modelers often want to talk about how close their model is to the **target output.** The most common way to do that is to compute the **Mean Square Error (MSE).** This is very simply defined. For any given input, compute the squared difference between each output node's activation and its target activation (by squaring this difference, we make each difference positive, so every node's error adds to the total error). Now take the mean of these numbers. That is the **MSE.** Since this only tells you how good the model is doing on a single **input pattern,** many modelers will compute the **MSE** for each member of the whole training set to see how the model is doing. **MSE** is also a nice way to determine how long to train the model—simply present inputs to the model and run your **learning rule** until **MSE** is below some arbitrary cutoff point.

Computing a single value for the performance of a **network** prompts many modelers to speak of the **error-space** or the **weight space.** Consider a network with only two **weights**. If we look at all the possible values for these weights and compute the **MSE** for each combination, we could plot a three dimensional **error-landscape** where the X axis was the first weight, the Z axis, the second weight, and the Y axis (vertical) the **MSE. Supervised learning algorithms** then simply search this **error-space** for the point (combination of weights) with the lowest

MSE. They start from a random point (remember our weights are set to random values initially) and wander until they can no longer reach a lower point. In doing this search, a model may fall into what is called a **local minimum.** A **local minimum** is simply a point in this **error space** that is lower than all of its neighbors, but may not be the *absolute* lowest point. Training the same model from several different starting points (random weight matrices) is a good way to escape this potential pitfall, as you are more likely to be sure that the final state is an **absolute minimum.**

A classic **back-propagation model** is the **autoassociator** (AKA the **autoassociative network**). This network is a three layer-network with the **same** number of **input** and **output nodes** and a **smaller** number of **hidden nodes** (thus the network is performing a **dimensionality reduction** as activation flows from **input** to **hidden nodes**). The network is trained to repeat whatever input it is given. This may seem trivial, but this is in fact an interesting problem given the dimensionality reduction.

For example, an autoassociator may represent a time-slice of a spectrogram by 100 nodes, but only have 4 hidden nodes through which to send that input to the output nodes. After computing hidden unit activations, it will need to recover 96 dimensions to go from hidden to outputs. In order to do this, of course, the **learning rule** must pick 4 dimensions to represent the input that are particularly important (account for a lot of the variance in the input). If this model is able to learn to perform its task, it may be very interesting what sorts of **hidden unit representations** it learns. In this particular task, we might expect the **hidden units** to approximate acoustic features.

Autoassociators bring up two very important concepts concerning **back-propagation** networks.

1) If you want to use your model to evaluate learning (ignoring for the moment issues about whether propagating the error signal is neurologically plausible), you must evaluate the **plausibility** of the **teaching signal**. It may be obvious that the **teaching signal** is doing a lot of the work in back-propagation networks. Since you could train a network to do virtually anything, given a good teaching signal, it is important to evaluate whether or not the signal you use is **psychologically** and/or **biologically plausible.** A word recognition network that is trained on acoustic input and told what the word is for each sound pattern is not very plausible, as real human babies don't generally have access to this. An autoassociator, however, does have a

plausible teaching signal, since brains probably do have access to their inputs. However, if you are not interested in learning itself, but rather, on whether or not a set of inputs are **learnable**, the plausibility of the teaching signal is not as much of an issue.

2) **Hidden unit representations** are important. In a lot of cases, (such as the autoassociator, they are the only interesting results. It is crucially important to evaluate what your hidden units are paying attention to in the input. This, of course, can often be difficult or even impossible, particularly in cases where the hidden units seem to represent inputs in arbitrary distributed representations. Often, however, individual hidden units will have some meaning that may be interesting. Evaluating what sorts of inputs the hidden units respond to can be very difficult. The best way is to treat the hidden units as a psychological experiment. Present them with various inputs that you have varied systematically to test one or more hypothesis. Then try to find out if the activation of certain hidden units (or groups of units) can be predicted by those hypotheses.

Unlike **supervised learning, unsupervised learning** requires no **target values** for the **output—**there is no right or wrong answer. Rather, **weights** are modified as a function of the **input** and **output activations** only.

One of the most common **unsupervised learning rules** is the **Hebb rule**, proposed by Donald Hebb in the late 1940s. Hebb (1948) actually proposed this rule long before we knew anything about **neural networks** (computational or biological) and it turns out to have been very useful in the computational literature and also has a close physiological correlate in a phenomena called **Long Term Potentiation** or **LTP** (that is to say that real neurons actually behave this way). Although some people use **unsupervised learning** and **Hebbian learning** synonymously, the strict definition of **Hebbian learning** states that if an **input node** and an **output node** are *simultaneously* active, the strength of their **connection** increases. For example:

$$W_{xy} = W_{xy} + I_x{*}O_y \qquad (7)$$

Here, if either I or O are equal to zero, there will be no change in weights. If they are both active, however, W will be increased. Since we can’t have weights increasing indefinitely, however, many modelers will include a **weight decay term** that says that if the nodes are not simultaneously active to decrease the weights. Of course we will also want to include a **learning rate** (which we will abbreviate as **ε**)

$$W_{xy} = W_{xy} + \varepsilon(I_x * O_y - W_{xy}) \qquad (8)$$

Here if I and O are active, we will increase W by a small amount (the old value of W multiplied by the learning rate). If they are not we will decrease it by a small amount.

Less common than **Hebbian Learning** is **AntiHebbian Learning** in which if an **input** and **output node** are simultaneously **active**, their connection decreases. Of course, there are many unnamed variants of these two **supervised learning rules**, but they are similar in that they do not depend on a **teaching signal.**

One common scheme for using **unsupervised learning** is **competitive learning** (or **winner-take-all learning,** see Rumelhart and Zipser (1986))**.** In this scheme before computing the **weight change,** the modeler sets the output node with the **highest activation** to one and all the others to zero. This is a simplification of a **lateral inhibition process.** Then the weights are changed according to a **Hebbian** or other **unsupervised rule.** The result of this sort of learning is that the model is able to find categories in the input (i.e. it will devote one output node to one category of inputs in the training set and a different output node to the others).

Another common scheme is the **Kohonen** (1982) **network (or Self Organizing Feature Map, SOFM).** A **Kohonen network** works very similarly to a **competitive learning network,** except that rather than exciting only the **winner** in the **output layer**, the **winner** and a number of it's **neighbors** are excited together, before applying the learning rule. The result of this is a distorted **map** of the input space in the **output space** in which regions of the **input space** that occur frequently in the training set have lots of **output nodes** devoted to them and other regions have fewer.

Hebbian learning has also been used in **Pattern Completion Networks** (famous examples are the **Brain-State-in-a-Box** and the **Hopfield Network**). These networks have only a **single** layer that serves as both the input and **output layers.** All of the **nodes** in this layer are connected to each other (**laterally)** and these **weights** are modified with Hebbian learning. The model is trained on a series of patterns until the weights settle. Then afterwards, the model can be given a partially complete pattern and will be able fill in the rest. For example, a four-node **pattern completion network** may be trained on the following **activation patterns**

[1 0 1 0]

[0 1 0 1]

With training, it will learn that when node #1 is on, node #3 should also be on, and that when node #2 is on, node #4 should also be on. So when presented with [1 0 _ 0], it will output the correct pattern, [1 0 *1* 0].

Noise

When building connectionist models, we can make them pure and pristine, perfect examples of what cognition *should be.* However, this is rarely a useful generalization since everything we know about the brain suggests it is as noisy as a debate on Chomskian language acquisition. To counter that objection, people often **add noise** to a **system**. This may seem abstract and weird but all they are doing is adding **small random numbers** to **something**. Sometimes this is added to the input layer before outputs are computed, sometimes it is added when the outputs are computed, and sometimes it is added to the weights. Using a **weight matrix** of small **random** numbers is another extremely common method of adding noise (although this is usually considered adding noise to the learning mechanism, without affecting the processing). Just know that adding **noise** is simply injecting a little **randomness** into the model somewhere.

Noise doesn't always **degrade** performance. Elman and Zipser (1988), for example, found that if they added noise to a speech recognition network it actually learned better, because the noise forced the network to create "noise-independent" representations of the speech. These representations were more useful in generalizing across speakers and contexts.

Another key point regarding **noise** is that once you add some to a **network,** your model is no longer **deterministic.** That is to say that every network is going to be slightly different (because you will be adding **different random numbers** to each instantiation). Because of this, you are not guaranteed that every network will be able to solve the problem, so it is a very good idea to run several different models under different noise conditions to determine how your model fares against noise. Conversely, when you read a paper in which noise is added to a model (even if it is just in the initial weight matrix), it is important to note whether the author ran the model several times. Otherwise, the possibility is open that he or she simply got lucky the first time (or didn't report the 200 models that failed). A network that generally solves the problem every time in differing levels of noise is said to be **robust against noise.**

Recurrence

Cognition often must unfold over time. In order for networks to capture this, **recurrence** is often added. **Recurrence** generally means that a **layer's activation** is in some way influenced by that **layer's activation** *at a previous time*. Some **recurrent networks** will have **layers** (such as an **output layer**) that are a function of themselves (at previous times). For example:

$$\text{Output}_{\text{time=t}} = \text{f}(\text{output}_{\text{time=t-1}},\text{inputs}\ldots) \qquad (9)$$

In the simplest case of this, the network may consist of only a single **layer** (which is both **input** and **output**) and simply connects to itself over time. The **Pattern Completion Network** discussed earlier is one such example. **Recurrent networks** usually take time to process a single input (as **activation** flows back and forth between nodes). Often, giving a **recurrent network** an input and allowing it to process it is called **running the network** (although this can often refer to training as well).

Other networks may have layers with more indirect influences on themselves. The TRACE model (McClelland and Elman, 1986), for example, is a type of **recurrent network** known as an **interactive activation model** (or **IAM**). In this model, activation starts at the feature level and is passed to the phoneme level and then to the word level. The word level then passes activation back down to the phoneme level (via **feedback**) connections, so that the phoneme activation at time 2 is a function of both the feature input and information from the word level (which of course is determined by the phoneme level at time 1). This process cycles over and over again through time and predicts a number of the results about the temporal dynamics of speech perception.

Another famous recurrent network is Elman's (1990) **simple recurrent network** (or **SRN).** These networks have been used to model all sorts of sequential behavior (of which language is probably the most interesting). They use back-propagation for learning and are trained to predict the **next input** they will receive. For example, if they are learning sequences of words such as "the dog smiles", and "the boy eats" at any one instance of "the" the **SRN** will be trained on the very next word (such as "dog"). Over time, this **SRN** should report that "boy" and "dog" are highly likely (active) after hearing "the", but "eats" and "smiles" are not.

Simple Recurrent Networks have a very simple structure that has turned out to be quite powerful. Activation starts in the **input layer** and flows into the **hidden**

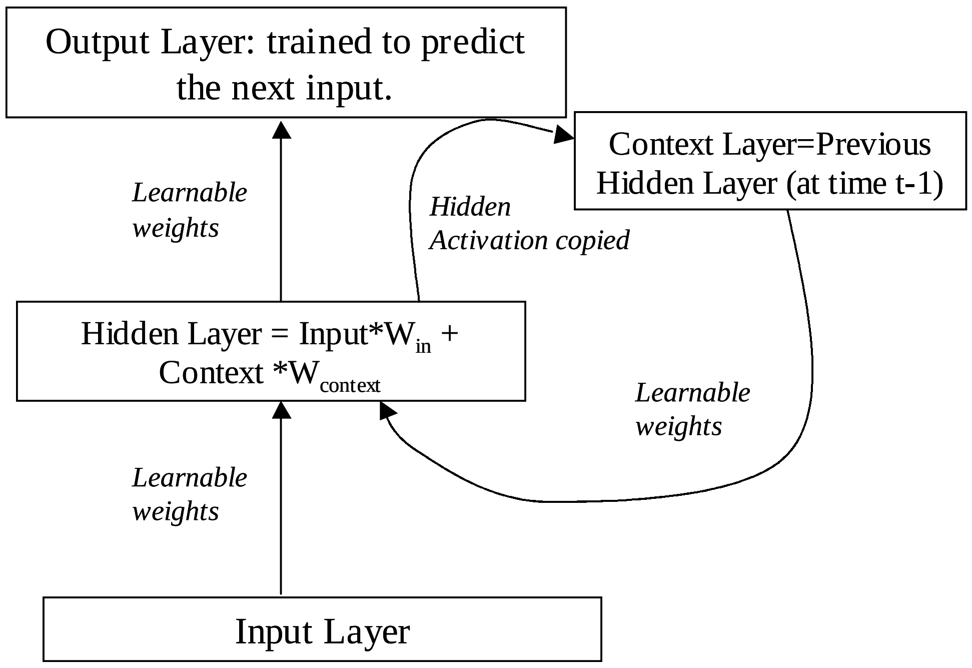
Output Layer: trained to predict the next input.
Learnable weights
Hidden Activation copied
Context Layer=Previous Hidden Layer (at time t-1)
Hidden Layer = Input*W_{in} + Context *$W_{context}$
Learnable weights
Learnable weights
Input Layer

Figure 5: A simple recurrent network. The network is trained to predict the next output. At each iteration, activation in the hidden layer is computed from the input and context layers. That activation is then copied to the context layer for the next iteration.

layer. Activation in the **hidden units** is not simply computed from the **input layer** alone, rather it is equal to the **input layer** multiplied by its weights *plus* the activation of the old hidden units (at the *last* time-step) multiplied by some other weights. **Output activation** is computed from these **hidden units**. Thus, when dealing with **temporal** stimuli (such as language), the **SRN** you will need to be basing outputs on not only the **current input** (for a word, for example, the current input might be a phoneme), but also on some of the **previous inputs** (the previous phonemes).

Although **SRN**'s are trained using ordinary **back propagation,** many other **recurrent networks** are trained using a **learning algorithm** called **back propagation through time.** In this algorithm, the network is literally unfolded over time so that the output layer at time 1 will be one physical layer of the network and the output layer at time 2 will be treated as an independent second layer of the network (see figure 6). The network can then trained as a regular old multi-layer network and the changes to all the weights (remember since each layer

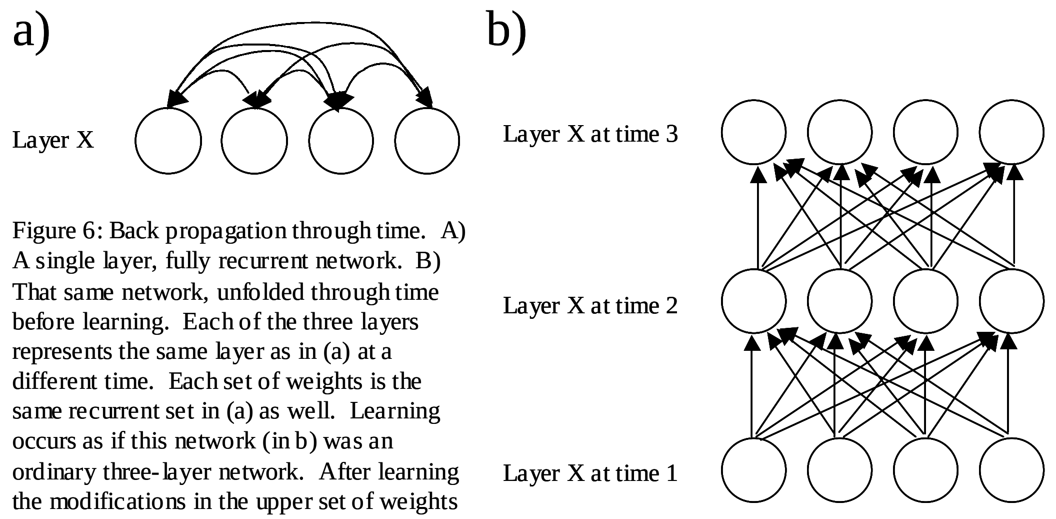

Figure 6: Back propagation through time. A) A single layer, fully recurrent network. B) That same network, unfolded through time before learning. Each of the three layers represents the same layer as in (a) at a different time. Each set of weights is the same recurrent set in (a) as well. Learning occurs as if this network (in b) was an ordinary three-layer network. After learning the modifications in the upper set of weights are combined with the modifications in the lower set of weights to change the final weight matrix.

of nodes really consists of the same nodes, each weight matrix is really the same weight matrix) will be combined to compute the final weight changes.

Genetic Algorithms

One emerging technique in connectionist modeling is the application of **genetic algorithms** to modeling. These algorithms seek to "breed" networks by using a technique reminiscent of **biological evolution**. Essentially, each network is assigned a **genome** that records its properties (such as the number of hidden units, the learning-rate, or the values of the weights). The most common scheme for encoding this genome is to use a string of **bits** (one or zeros). Each group of bits or **gene** (maybe the first 10, for example) will encode (in base-2) the value of whatever parameter that **gene** represents.

Once the form of the **genome** is determined, a large number of networks will be generated by creating **genomes** at **random.** These networks are all trained, and after they have all been run their **fitness function** is evaluated. This function essentially tells the algorithm how good the network did at accomplishing its task. The next **generation** of networks is then created by combining the genomes of the networks with highest **fitness values.** Sometimes **mutations** are allowed to creep in by randomly changing one or more bits of the genome. There are literally thousands of different mechanisms for evaluating fitness, organizing the genome, computing the genomes of the next generations, and having mutation. I direct the reader to Mitchell (1999) for a good introduction to them.

There is nothing mathematically special about **genetic algorithms.** They simply form another class of search tools for fitting a model to a data. Other classes include learning rules like **Back Propagation** or statistical optimization techniques like **Maximum Likelihood Estimation.** The reader should bear in mind that among these optimization tools, **genetic algorithms** are the most poorly understood, and may not be the most efficient (they will take longer to solve the problem than other techniques).

Genetic algorithms are popular mostly because of the compelling (to some people) **biological analogy** they provide. However, a close look at this analogy suggests they may not be as compelling as many people think. Researchers have used **genetic algorithms** to set the weights of a network as well as to determine features of the architecture (number of nodes, connectivity, learning rule, etc..). However, if you accept the majority-view that **weights** encode *learned knowledge,* it is hard to accept the evolutionary analogy for genetically determined weights as we have yet to find evidence for inherited knowledge. Moreover when **genetic algorithms** are used to determine the architecture of a model it is often extremely difficult to understand how a model is solving a particular task and how the **genetic algorithm** arrived at that solution. Because of this, such models are not good instantiations of a theory—since the theorist did not determine how the model processes information, "evolution" did—unless your theory is a theory about evolution (and then you run into the problem that the model of evolution in most **genetic models** is quite bare). I am not trying to say here that **Genetic algorithms** are useless. They do have their place in **connectionism,** but we must exercise caution in building them (and reading about them) to be sure that we are saying something interesting, interpretable and new about cognition. To really achieve any utility we must constrain the algorithms to the point where we can understand the output.

Damage and Lesions

A growing body of literature has begun to examine what happens when a network is **damaged**. This has been particularly fruitful in language research as it is often useful to compare the output **lesioned network** with that of an **aphasic.** Much like the use of **noise** in connectionist networks, this **lesioning** a network is a concept that is much less complicated that it might seem.

Researchers have come up with two major ways of damaging a network. The first is simply to remove some **connections** (**weights**) between nodes by setting them to zero permanently. The second is to remove one or more nodes (typically **hidden units**). In both cases, people have looked at damaged networks in two ways. Often they will simply compare their performance after the damage with real data from patients. Other times, after receiving the damage, the network will undergo some more training as a simulation of recovery. This is particularly interesting in the case where hidden units are lost (in an **autoassociator**, for example) as this asks the question of whether the network can successfully adapt to having fewer dimensions with which to represent its inputs.

As I mentioned previously, the way in which **damage** is dealt with is one way in which **localist** and **distributed representation** schemes differ since **distributed representations** can deal with it more gracefully. Most networks exploring the effects of damage use distributed representations for this reason.

Discussion

I'll prewarn the reader that as I attempt to sum-up this article, my discussion is likely to turn into a personal pulpit for how connectionism should be done right. Other authors disagree with me of course, as many of these issues are either under active debate and those that aren't have simply not yet surfaced as dominant issues in the literature (although I predict that they may soon).

Connectionism has rapidly become a dominant tool for expressing and quantitatively modeling theories about psychological and neurological phenomena. Its use is growing in linguistics and it is our hope (on the psychological side of the fence) that more linguists will begin to add it to their theory building toolboxes.

It has been shown that given enough **hidden units** and enough layers of **hidden units, back-propagation networks** can learn to solve **any** problem (whether or not they can help my love life is a different story…). As a result of this, when evaluating network models we need to determine a lot more than whether or not the model does the task, but also things like

1) Is the structure of the model neurologically plausible? Does the model perform computations that real neurons could not possibly do?
2) Are the posited input and output representations psychologically and neurologically plausible? A model that builds syntactic trees and is given parts of speech may not be all that interesting (unless we have a

good model of part-of-speech tagging), since it is unlikely the syntactic processor is simply given these…

3) What feature of the model allows it to solve the problem? How does it solve it?
4) Does the time-course over learning and/or processing match the same time-course in humans?
5) And most importantly, what is the **linking hypothesis** between the model and the data? Models do not output eye-movements, or button-presses or EEG waves or grammaticality judgments or reaction times. Whenever we relate model output to actual data, we must form some **linking hypothesis** as to how this relationship holds. It is crucial that this be made explicit and that it be well reasoned. Additionally, this **linking hypothesis** is just as important a part of theory building as the model itself: the same model with different **linking hypotheses** can often yield strikingly different results.

When building a model, one needs to keep similar issues in mind. Although there is a large engineering literature that focuses on building models with the single goal of solving a particular problem, for the most part, connectionist networks in psycholinguistics and linguistics are built to instantiate a theory of language processing or learning (or some other aspect of language). In these models, there are a number of decisions to be made, and the best modelers will make these decisions on the basis of the theory they are trying to instantiate.

1) **Localist** or **distributed representation**? If a goal is neurological plausibility, distributed representations may be preferred (as **grandmother cells** have not yet been found in the brain) however a **topographic map** may be even better. If the goal is to relate output to discrete experimental responses, then maybe a **localist representation** will make it easier to do that.
2) What is the goal of learning? If you wish to model the time course of development or acquisition, maybe a more neurologically plausible **unsupervised** scheme is best. However, if you merely wish to show that a particular categorization or mapping is learnable from the input, a **supervised learning rule** may suffice. This distinction is not very clear-cut in the literature (many developmental arguments have been made with **back-propagation**), but it is important to keep in mind when building the model. If you do use a **supervised learning rule**¸ what is the basis of the **teaching signal**? Could it arise in real life with real brains/minds? Maybe you aren't interested in learning at all, but rather, are more interested in exploring processing mechanisms. Here

you may even consider setting the weights manually, or with a **genetic algorithm**.

3) Are you striving for a completely neurologically plausible architecture or is an abstraction enough? The answer to this can often constrain all the architectural choices you might need to make.

Because of the power inherent in **connectionist networks** and because they are often as opaque as the cognitive system they are attempting to model, several cautions must be exercised. Models must be developed to implement specific theories, and a specific **linking hypothesis** must be formed linking the **model** with the **data**. The architecture of the model should be grounded in good linguistic and psychological theory and should be tied to the theory we wish to instantiate. We should make every attempt to understand **how** a network solves the task, not just that it solves it, constraining our architectures toward this end if that is necessary.

Finally we should systematically explore the models we develop in a style similar to that of good psychological experimentation. We should always compare multiple instantiations of the same model. The effect of different sources and levels of noise should be systematically explored. Modelers should test the architecture of the model by looking at the effects of individual components of the network (e.g. running a network both with and without lateral inhibition). Lastly, models should be developed so that they can be directly compared to other models of the same phenomena. In the long run, only by combining these cautions with knowledge of the neuroscience, mathematics and psychology behind **connectionist modeling** will it ultimately prove useful as a tool for conceptual understanding and theory testing.

References

Elman, J., Bates, E., Johnson, M., Karmiloff-Smith, A., Parisi, D., and Plunkett, K. (1996) *Rethinking Innateness: a Connectionist Perspective on Development.* Cambridge, Mass.: The MIT Press.

Elman, (1990) Finding structure in time, *Cognitive Science*, *14*, 179-211.

Elman, J. and Zipser, D. (1988) Learning the hidden structure of speech. *The Journal of the Acoustical Society of America, 83(4),* 1615-1626.

Hebb, D. (1948) *The Organization of Behavior*. New York: Wiley

Kohonen, (1982) Self-organized formation of topologically correct feature maps. *Biological Cybernetics, 43,* 59-69

McClelland, J., and Elman, J., (1986) The TRACE model of speech perception. *Cognitive Psychology, 18,* 1-86.

McClelland J., Rumelhart, D., eds. (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 2,* Cambridge, MA: the MIT Press.

Rumelhart, D., Hinton, G., and Williams R. (1986) Learning internal representations by error propagation. in Rumelhart, D., McClelland, J. (eds.) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1.* Cambridge, MA: The MIT Press. 151-193

Rumelhart, D., McClelland J., eds. (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1,* Cambridge, MA: the MIT Press.

Rumelhart, D., and Zipser, D. (1986) Feature discovery by competitive learning. in Rumelhart, D., McClelland, J. (eds.) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1.* Cambridge, MA: The MIT Press. 151-193

Acknowledgements

This paper grew out of a handout for a series of discussions on the relationship between linguistics, psychology and connectionist models in speech perception that included Mike Tanenhaus, Mikhail Masharov, Jim Magnuson, Katherine Crosswhite and Joyce McDonough. As a group they've contributed to this paper by introducing me to linguistic thinking, and forcing me to think about connectionism from this viewpoint. I'd like to thank Katherine in particular for helping to make this readable to linguists and Robbie Jacobs for helping to make this readable to connectionists by keeping my facts factual.