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.

In any model, one or more **layers** is designated the

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).

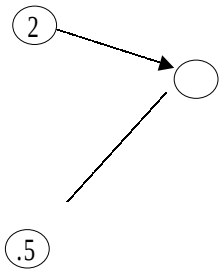
In a network **nodes** are connected to each other by **weights** (AKA **synapses**, **connections**). Each **weight** represents the amount of **activation** that can be passed by one **node** to another. If an **input node** is highly active and it has a strong **connection** to an **output node**, 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.



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.**

$$output_{top} = f(\ input_{top}, input_{bottom}, weight_{top->top}, weight_{bottom->top}) \qquad (1)$$

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.

$$output_{top} = input_{top}*weight_{top->top} + input_{bottom}*weight_{bottom->top} \qquad (2)$$
$$output_{bottom} = input_{top} * weight_{top->bottom} + input_{bottom} * weight_{bottom->bottom}$$

This can be generalized to:

$$output_y = \sum_{x=1}^{Num\ input} input_x * weight_{x->y} \qquad (3)$$

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.

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

$$W = [\text{weight}_{top\text{->}top}$$

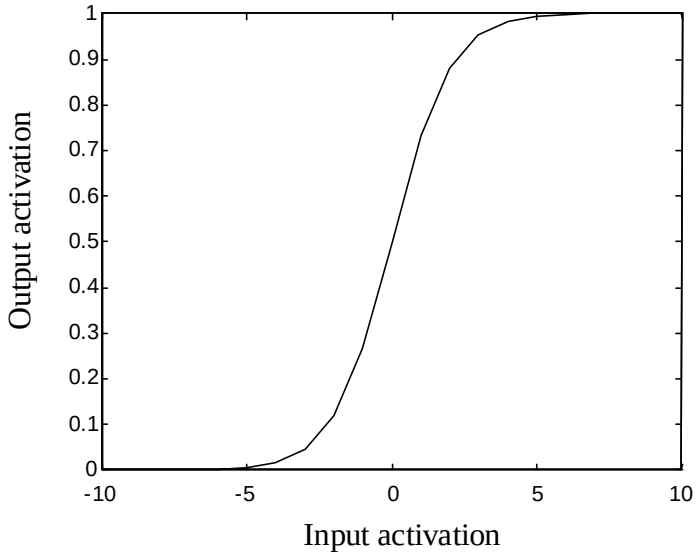
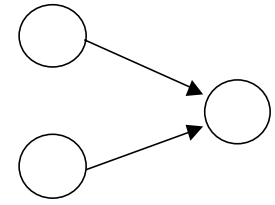


Figure 3: The logistic activation function. For any input activation to an output node, the logistic function outputs a value between 0 and 1.

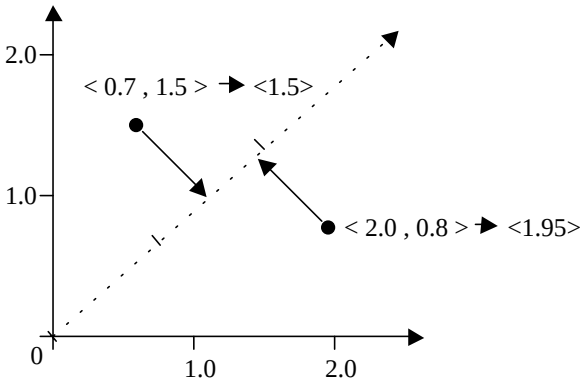
logistic activation function serves to truncate the possible values of the output activation to a value between 0 and 1. If the sum of inputs*weights is high, the output node will equal 1. If that sum is low, the output node will have an activation of 0.

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

logistic



Input Output
[2.0 0.8] * W = 1.95



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?"

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

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 correcte

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

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

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

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

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:

$$Output_{time=t} = f(output_{time=t-1}, inputs\ldots) \quad (9)$$

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 "the24 at,-0. in,i345 -1.110.001 Tt(,-0.)TjıT*ıi*3Se42

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

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

o r a 9F / 1 t e

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