4 Neural Networks and Backpropagation

The human brain consists of about 100 million neurons. Neurons typically consist of three parts: the cell body or **soma**; various **dendrites** which feed into the cell body; and an **axon** which terminates with a branching called the **axonal arborization**, at the end of which are the **axon terminals**. Certain other cells, called **Schwann cells** wrap themselves around the axon, forming its **myelin sheath**.

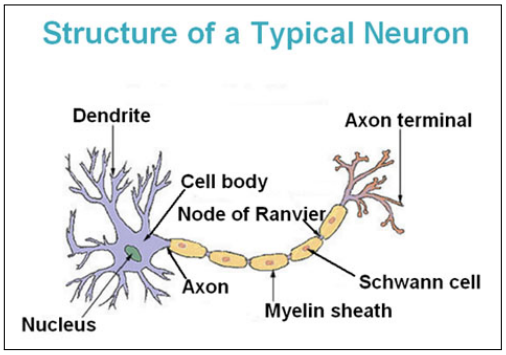

Figure 19 A Neuron. From Wikipedia.

A neuron A connects to another neuron B by attaching an axonal branch to the dendrite of neuron B. The connection between the axonal branch and the dendrite is known as a synapse.

Neurons send electrical signals to one another. However, neurons are not actually electrically connected at the synapses. Instead, neuron A will send an electrical signal down its axon to the tips of the axonal arborization, where the synapses are located. At the synapse, this electrical signal causes the axon to release a chemical, called a **neurotransmitter**, which crosses the synaptic junction to a dendrite of neuron B. This neurotransmitter excites the dendrite which causes it to slightly increase or decrease the voltage inside neuron B. A synapse with neurotransmitters which *increase* the voltage of the receiving neuron is called an **activating synapse**. A synapse with neurotransmitters which *decrease* the voltage is called an **inhibitory synapse**.

When the voltage of a neuron has increased beyond a certain level (due to repeated excitation through incoming synapses), the neuron "fires" a signal pulse down its axon and resets its voltage back to normal values. This firing is known as an action potential, and it is what is responsible for the neuron sending signals to other neurons. After it fires, the neuron goes into a sleep mode temporarily, where its voltage cannot be increased no matter how many incoming synapses are excited. This waiting period is known as the **refractory period**.

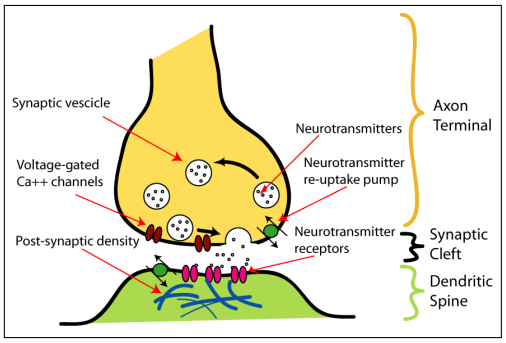

Figure 20 A Synapse. From Wikipedia.

A common refractory period is about 40ms. This means that such a neuron cannot possibly fire more than about 25 times a second. Multiply this by 100 million, and you get an absolute (and totally unrealistic) maximum of 2.5 billion firings a second in the brain. That's 2.5 gigahertz. With two 1.5 gigahertz Athalon processors, we can now purchase a $5K computer which operates faster than the maximum throughput of the human brain. So why don't we have AI yet? One reason is that neurons have a large branching factor, both in through the dendrites and out through the synapses. In a large number of neurons in the brain, the branching factor approaches 10,000 per neuron. Thus while we might have 100 million neurons or so, we have many many billions of synapses. So we have a ways to go.

The synapses are a vital part of the brain, and not just in hooking neurons to neurons. Synapses are also believed to be the primary mechanism for memory in the brain. Memory information is stored by modifying the "strength" of a synapse, that is, its ability to increase or decrease the voltage of the receiving neuron.

Neurons come in three general flavors:

- **Sensory Neurons** do not receive signals from neurons through their dendrites. Instead, these neurons are customized to respond to sensory input. For example, the cones and rods in the eye are sensory neurons, as are taste buds and olfactory (smell) or touch-sensing neurons.
- **Motor Neurons** do not send signals to other neurons via their axons. Instead, their axons control muscle tissue.
- **Interneurons** receive signals from other neurons through their dendrites, and send signals to other neurons through their axons.

The notion that we might be able to model thinking by modeling neurons in a computer (or electronics) is a very old one. The first neural networks were conceived in 1943 by McCulloch and Pitts.

4.1 McCulloch and Pitts Neurons

McCulloch and Pitts neurons are very simple. At any point in time, a neuron x_i adds up the number of firing incoming neurons x_j, each multiplied by a different synapse weight w_{ij}. If the amount exceeds a threshold u_i, then the neuron fires. When a neuron is firing, its value is 1. Otherwise, it is 0. A neuron at time t fires according to the following equation based on the previous values of the neurons at time $t-1$.

$$x_i^{(t+1)} = \text{step}(\sum_j w_{ij} x_j^{(t)} - u_i)$$

$$\text{step}(u) = \begin{cases} 1 & u \geq 0 \\ 0 & u < 0 \end{cases}$$

Or in matrix notation:

$$\vec{x}^{(t+1)} = \text{step}(W\vec{x}^{(t)} - \vec{u})$$

For McCulloch and Pitts neurons, it's the job of the neural-network designer to come up with the number of neurons, the topology of the synaptic connections, appropriate synapse weights and thresholds for each neuron in order to get the neural network to do the task desired. If you pick the right values, a graph of McCulloch and Pitts neurons can provably compute anything. However, McCulloch and Pitts neurons don't do the one thing which neural networks are most famous for: **learn**. To do that, you need an automated procedure for modifying these variables.

4.2 Perceptrons

The first major such approach was the **perceptron**, a simple neural network first studied by Rosenblatt in 1962. The most basic kind of perceptron consists of sensory neurons which are synaptically connected to motor neurons; in other words, input is fed into the sensory neurons, and the output of the network is registered by the motor neurons. Here's a typical design perceptron:

This picture is very typical of neural networks. Neurons (the nodes) are linked via edges which represent the synaptic connection between two neurons. Each synaptic connection has a particular **weight** associated with it, which indicates the strength of the synapse. The edges are directed: the head of the edge indicates the neuron receiving the synapse through its dendrites. There can be any number of input and output neurons—it just depends on the problem at hand.

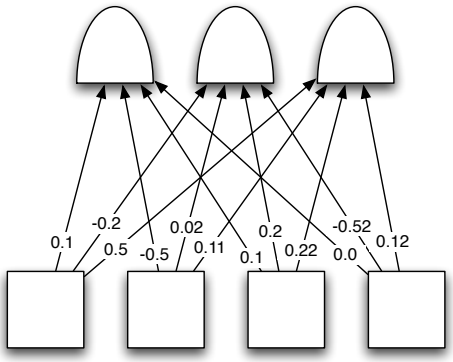

Figure 21 A 4-Input, 3-Output Perceptron.

A neural network of this type is called a **feed-forward** network, meaning that there are no cycles in the graph. The sensory neurons fire according to their inputs, and the motor neurons eventually fire an output. The basic perceptron is a **one layer** feed-forward neural network, meaning that there is only one layer of synapses and two layers of neurons. We'll discuss multilayer networks later.

The weights of a perceptron start at 0. Perceptrons are supposed to learn the correct output for a given input. Inputs and outputs are vectors. The input to a perceptron is a vector of 1's and 0's, one per input neuron. The output is also a vector of 1's and 0's, one per output neuron. The user sets the value of each input neuron to the corresponding item in the input vector (a 1 or 0). The value of an output neuron i is computed according to this equation:

$$\vec{o}_i = \text{step}(\sum_j W_{ij}\vec{x}_j)$$

Or in matrix notation,

$$\vec{o} = \text{step}(W\vec{x})$$

$\vec{o}_i$ is the value of output neuron number i. $\vec{x}_j$ is the value of input neuron number j. W_{ij} is the weight of the synapse (connection) between them.

The user then checks the output values against the desired output values. To get the neuron to learn to produce the desired output values, the weights are then modified according to a learning rule. The original (now defunct) learning rule used for Perceptrons was as follows:

1. If output neuron $\vec{o}_i$ incorrectly produced a 1, then we need to decrease W for i. For each incoming synapse to $\vec{o}_i$, set $W_{ij} = W_{ij} - \vec{x}_j$
2. Else if output neuron $\vec{o}_i$ incorrectly produced a 0, then We need to increase W for i. For each incoming synapse to $\vec{o}_i$, set $W_{ij} = W_{ij} + \vec{x}_j$
3. Else if output neuron $\vec{o}_i$ got the right value, then make no changes to its weights.

This rule is really a special case of what I think is a better, simpler, and more general learning rule, which looks like this:

$$\Delta W_{ij} = \alpha(\vec{y}_i - \vec{o}_i)\vec{x}_j \tag{1}$$

Or in matrix notation,

$$\Delta W = \alpha(\vec{y} - \vec{o})\vec{x}^T$$

Note the outer product of $\vec{y} - \vec{o}$ being multiplied against $\vec{x}^T$.

$\vec{y}_i$ is the "correct" (desired) output value for neuron i. α is the **learning rate**. If the learning rate is high, then the neural network will learn more rapidly, but if it's too high, then the network can bounce around and never settle down on the solution. If the learning rate is small, then the network will take a long time to learn, but it may nail the solution more exactly. A learning rate of 0.1 is not uncommon, but you'll have to adjust it depending on your problem.

The overall training procedure for a perceptron thus works as follows. Given a set of input vectors and corresponding expected output vectors for a function,

1. Pick an input/expected-output vector pair at random.
2. Present the input vector pair to the input neurons.
3. Read the network's output vector.
4. Using the learning rule, modify each weight according to the difference between output vector and the expected-output vector.
5. Go to #1.

The procedure finishes when the network is reliably producing the expected output for every input.

Perceptrons of this form can learn some but not all boolean functions. There are two reasons for this. First, the output of the perceptron is 1 only if the inputs sum to ≥ 0. What if we want the output of the perceptron to output $\vec{o}$ when this is the case? Presently we cannot do this.

The reason for this is related to the following: imagine an equation of the form $Ax + By = 0$. It doesn't matter what A or B are: this equation must pass through $(0, 0)$. There's no way to set A or B to move the line so that it passes through a different y or x intercept. So there's a limitation on the kinds of lines we can draw. In order to fix this, you need to add a bias C, so you get the equation $Ax + By + C = 0$.

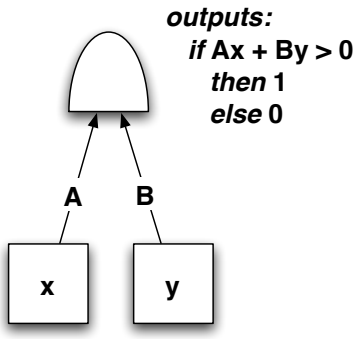

Figure 22 A Simple Perceptron.

So imagine a simple perceptron with just two inputs and one output, such as the one in Figure 22 at right. Here we've labeled the two inputs x and y, and the two weights A and B. If the weighted sum is ≥ 0, then the output is 1, else it is 0. Thus the equation which divides the space up between the "1"s and the "0"s is... $Ax + By = 0$. Thus this perceptron can only draw certain dividing-lines, not all of them. What we need is the ability to add a "C" into the perceptron. We do that with a bias unit, as shown in Figure 23.

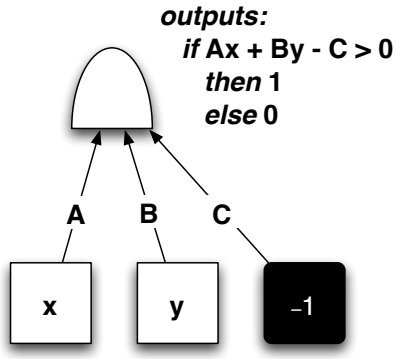

Figure 23 A Perceptron with a Bias Unit.

A bias unit is an input neuron which always fires the same constant value, regardless of the input. In this example, our bias value is always firing a -1. It can be something else if you like. This fixed-value neuron, and its weight C, is now is

included in the output neuron's equation, and we modify the weight C just like the other weights. Now the perceptron can create any dividing line it needs.

Before we added the bias unit, our neural network was capable of learning only certain boolean functions. With the bias unit, we have extended the learning capacity to include functions like AND and OR and NAND and NOR, which couldn't be done with the previous network. This is because the dividing line with the bias unit is now $Ax + By + C(-1) = 0$. We now can have a non-zero Y-intercept. And more significantly, ***C* is the same as u_i in the McCulloch and Pitts neurons:** it's the value of the threshold that $Ax + By$ must exceed.

Even with a bias unit, there are still boolean functions which can't be learned. For example, consider the six standard boolean functions below:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

NAND	0	1
0	1	1
1	1	0

NOR	0	1
0	1	0
1	0	0

EQ	0	1
0	1	0
1	0	1

XOR	0	1
0	0	1
1	1	0

Notice that in each of the first four functions, it's possible to draw a straight dividing line which separates the 1's from the 0's in the 2×2 box regions. But for XOR, there's no way to divide the 1's from the 0's with a single straight dividing line. No matter what line you draw, you'll either have 0's on both sides or 1's on both sides, or both on both sides.[52]

Since you can't draw a dividing line, you can't make it such that our simple perceptron always outputs 1's when given (0,1) and (1,0) and always outputs 0's when given (0,0) and (1,1). The perceptron cannot learn XOR. (Likewise for EQ).

In general, perceptrons of any form can only learn boolean functions which are linearly separable. For a 2-variable input, this means functions where the 1's and 0's can be divided up with a straight line. For a 3-variable input, this means functions where the 1's and 0's can be divided up with a plane. And so on for higher-order inputs.

It was the perceptron's restriction to linear separability that doomed it. In a paper by Minsky and Papert, they presented this unfortunate fact, but they also predicted that any more "advanced" forms of perceptrons (with multiple layers for example) were also unlikely to escape the problem of linear separability. Minsky was held in such wide regard that this effectively wiped out the entire field of neural networks for over a decade. But Minsky and Papert were wrong! In fact, not only can multilayer neural networks learn more advanced functions than just linear separable ones, they can learn any function.

4.3 Error Backpropagation

You might notice that the Perceptron was used to divide space into (in this case two) **classes**. Thus the perceptron is a **classification algorithm**. But neural networks are actually more general than this. They can learn a continuous function from one multidimensional space to another: that is, they work well as **regression algorithms**.

Before we show this, let's generalize the single-layer perceptron in two ways. First, we will change the **step** function to a function with a more gradual change from 0 to 1; this will retain the nonlinearity of the function but the new function will be differentiable. Second, we define a formal learning algorithm in terms of **gradient descent**. At that point, we will **still have a glorified**

[52] In fact, none of these six functions can be learned without a bias unit, because the only functions that can be learned without a bias unit are ones which define lines which come out of the origin (the intersection of the horizontal and vertical lines in each table). So, which booleans *can* be learned without a bias unit?

linear-separability learner, just one which learns a soft function rather than a hard line. Only in the next section will we graduate to a full function learner.

The Sigmoid Function We begin by replacing the **step** function with a function called **sigmoid** (or σ). In fact we can use any nonlinear differentiable function, but σ is convenient because it has an unusual derivative in terms of itself.[53] σ is defined like this:

$$\sigma(u) = \frac{1}{1 + e^{-\beta u}}$$

The value of β affects the slope of the curve, and thus the "softness" of the differentiation. We'll assume $\beta = 1$, which is typical. Sigmoid (with $\beta = 1$) is shown in Figure 24, and Figure 26 shows Sigmoid for various values of β.

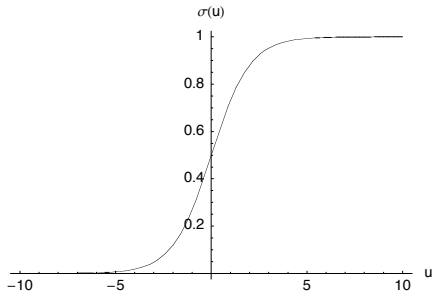

Figure 24 The Sigmoid function with $\beta = 1$.

What makes sigmoid nifty is its derivative:

$$\sigma'(u) = \sigma(u)(1 - \sigma(u))du$$

Huh. Sigmoid's derivative is recursive. Cool. That will make the math happy. Another common function to use is hyperbolic tangent, which looks very similar except that it ranges from -1 to 1. It also has similar differentiation properties as sigmoid.

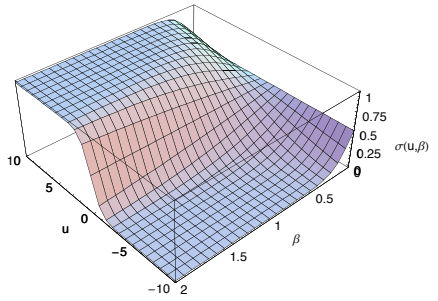

Figure 25 The Sigmoid function with β ranging from 0 to 2.

Our new perceptron learns real-valued functions, not just boolean functions. This means that the neurons in the network can output any real number. To handle this, we need a new function describing the output value of a neuron. So now $\vec{o}_i$ is defined as:

$$\vec{o}_i = \sigma(\sum_j W_{ij}\vec{x}_j)$$

Or in matrix notation,

$$\vec{o} = \sigma(W\vec{x})$$

Unlike in the perceptron, the initial weights of a multilayer neural network start with small random values centered around 0—perhaps random values between -.01 and .01, for example. This is because if the weights are all 0, the network cannot learn—the weights must be non-zero to learn.

Training To train the neural network, we present a random input and determine the values of the output layer. We compare the results of the output layer to the correct results we're trying to train the network to produce. Then we modify the weights in W so that they are closer t producing the output. The rule we use for modifying the weights is known as the delta rule, because it changes each weight according to how much say it had in the final outcome (the delta, or partial derivative of the output with respect to the weight). Before we derive the delta rule, here's the rule itself:

[53] Another popular choice is hyperbolic tangent, or $\tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$, which looks similar to sigmoid and has similar related first-derivative properties. Specifically, $\tanh'(u) = 1 - \tanh(u)^2$.

$$\Delta W_{ij} = \alpha(\vec{y}_i - \vec{o}_i)\vec{o}_i(1 - \vec{o}_i)\vec{x}_j$$

Or in matrix notation,

$$\Delta W = \alpha((\vec{y} - \vec{o}) \times \vec{o} \times (1 - \vec{o}))\vec{x}^T$$

...where $\times$ is an "element-by-element multiply": multiply each element in matrix A by the corresponding element in matrix B to produce a third matrix (the three matrices must be the same dimensions). Also, $1 - \vec{o}$ subtracts each element in $\vec{o}$ from the scalar 1. Note again the outer product using $\vec{x}^T$.

Notice that the learning rule for W_{ij} is very similar to that of the standard perceptron. The difference is the inclusion of $\vec{o}_i(1 - \vec{o}_i)$. Where was this in the perceptron learning rule? The perceptron learning rule used the Step function, and not the Sigmoid function; the Step function does not have a derivative, where the Sigmoid function does have a derivative, and as the proof below will show, $\vec{o}_i(1 - \vec{o}_i)$ comes from the derivative. Plus, if you think about it, $\vec{o}_i$ was always either 1 or 0 for the Perceptron, so $\vec{o}_i(1 - \vec{o}_i) = 0$. The Perceptron wouldn't learn at all with this term!

The overall procedure for training a backpropagation neural network is similar to the perceptron procedure. Given a set of input vectors and corresponding expected output vectors for a function,

1. Pick an input/expected output vector pair.
2. Present the input vector pair to the input neurons.
3. Let the values flow through the neurons up to the output neurons.
4. Read the network's output vector.
5. Using the delta learning rule, modify each weight according to the difference between output vector and the expected-output vector.
6. Go to #1.

You could pick pairs at random with replacement; or you could shuffle all the pairs and then iterate through them, then re-shuffle and re-iterate, etc. What I *wouldn't* do is repeatedly iterate through your pairs without shuffling every once in a while.

The procedure finishes when the network is reliably producing something very close to the expected output for every input we provide it. Remember, this network operates on continuous values, so we can't nail the expected output exactly. What we will aim for is that the network will converge to the correct output at the limit.

Just like in a regular perceptron, sometimes a backpropagation network will converge for some inputs but fail to converge for others. This is because the network got caught in a suboptimum solution and can't find its way out of it and on to the global optimum. Backpropagation has this property because it is a greedy algorithm, and so it is not guaranteed to work; you may have to run it many times to get all the inputs to converge.

One way to help the neural network converge is to lower the learning rate; it'll take longer to learn, but will have a better chance of finding the global optimum. Another way to help it is to increase the number of neurons in the hidden layer. However, this second approach has a

downside: if you increase the neurons in the hidden layer by too much, then the network will learn exactly the inputs you provide it, but won't be able to come up with a general solution. Often you can't provide all the inputs to a network because there are just too many. In this case, you want the network to learn the function from just a subset, and successfully generalize what it learned to all possible inputs as a whole. Large numbers of hidden layer neurons make this less likely to succeed.

Generalization is a good thing! In our brains, we generalize all the time; it's the only way we can possibly come up with guiding rules based on just the limited inputs we receive through our senses. However, generalization also has some negative social results, such as stereotyping.

Proof of the Delta Rule We finish up with a proof of the delta rule. The idea behind the delta rule is as follows: each weight contributes some amount to the output of the network. We want to change each weight so that its contribution will help the network more closely approximate the output we had expected. The gradient of the output of the network is the change in the output with respect to the change in weights—that is, it's the first derivative with respect to those weights. We want the network to go away from the direction the gradient is pointing (to make the difference smaller). This is the general idea behind gradient descent.

We measure the difference between the output and the desired output with an error metric. Our goal will be to move in the direction that makes this error metric as small as possible (away from the error gradient). The error metric we'll use is a mean squared error $\mathcal{E}$ between the output and our desired correct output:

$$\mathcal{E} = \frac{1}{2}\sum_i (\vec{y}_i - \vec{o}_i)^2$$

The rule for an error metric is that (1) it has to be zero for zero error, and (2) larger errors need to be more positive. Other than that, it can be anything we want. Clearly $\sum_i (\vec{y}_i - \vec{o}_i)^2$ fits this bill. We add the $\frac{1}{2}$ because it'll make the math prettier.

We're going to move each weight independently down the error gradient by a little bit (by α). The gradient with respect to a single weight is the same thing as the partial derivative of the output with respect to that weight. For a given weight W_{ij}, we want the change to be (expanding out with the chain rule):

$$\begin{aligned}\Delta W_{ij} &= -\alpha \frac{\partial \mathcal{E}}{\partial W_{ij}} \\ &= -\alpha \frac{\partial \mathcal{E}}{\partial \vec{o}_i}\frac{\partial \vec{o}_i}{\partial W_{ij}}\end{aligned}$$

Remember, it's negative because we want to move away from the gradient. Using the error function, the first term reduces to:

$$\frac{\partial \mathcal{E}}{\partial \vec{o}_i} = (\vec{y}_i - \vec{o}_i)(-1)$$

For the second term, we a function which has $\vec{o}_i$ and W_{ij} in it. That function would be the forward-propagation equation, $\vec{o}_i = \sigma(\sum_j W_{ij}\vec{x}_j)$. Using the nifty derivative property of sigmoid, we have:

$$\frac{\partial \vec{o}_i}{\partial W_{ij}} = \sigma(\sum_j W_{ij}\vec{x}_j)(1 - \sigma(\sum_j W_{ij}\vec{x}_j))\frac{\partial \sum_j W_{ij}\vec{x}_j}{\partial W_{ij}}$$

Notice that, due to the magic of sigmoid's differentiation, the first term is just $\vec{o}_i$. And the second term is just $1 - \vec{o}_i$. So now we have

$$\frac{\partial \vec{o}_i}{\partial W_{ij}} = \vec{o}_i(1 - \vec{o}_i)\frac{\partial \sum_j W_{ij}\vec{x}_j}{\partial W_{ij}}$$

To solve the last term, keep in mind that we're not finding the partial with regard to any W_{ij}. We're finding it with regard to a *specific* W_{ij}. All the other W_{ij}s are thus held constant. Thus the partial derivative reduces to just (a specific) $\vec{x}_j$. So we have

$$\frac{\partial \vec{o}_i}{\partial W_{ij}} = \vec{o}_i(1 - \vec{o}_i)\vec{x}_j$$

Putting this all together, we have:

$$\begin{aligned}
\Delta W_{ij} &= -\alpha\frac{\partial \mathcal{E}}{\partial W_{ij}} \\
&= -\alpha\frac{\partial \mathcal{E}}{\partial \vec{o}_i}\frac{\partial \vec{o}_i}{\partial W_{ij}} \\
&= -\alpha(\vec{y}_i - \vec{o}_i)(-1)\vec{o}_i(1 - \vec{o}_i)\vec{x}_j \\
&= \alpha(\vec{y}_i - \vec{o}_i)\vec{o}_i(1 - \vec{o}_i)\vec{x}_j
\end{aligned}$$

We could have done this with any differentiable nonlinear function, but sigmoid makes it pretty because all the nastiness drops out when you can substitute it in for itself at the end like that.

4.4 Multi-Layer Neural Networks

So far, even with a nonlinear function, we just have a glorified linear discriminator with a soft margin. What Minsky and Papert didn't see was that if you *add layers*, you can convert this exact same system into a full-blown function-learner. You can learn any continuous, differentiable function mapping one multidimensional space to another.

To do this, we're going to add a single additional layer. The network now looks like this:

This network is like a perceptron, except that it has an additional added layer of interneurons, called a hidden layer. The number of neurons in the input and output layers is dictated by the size of the inputs and outputs for the problem we're trying to learn. But we can make the hidden layer any size we like.

The values of neurons in the output layer are (still) designated with the column vector $\vec{o}$, and each element

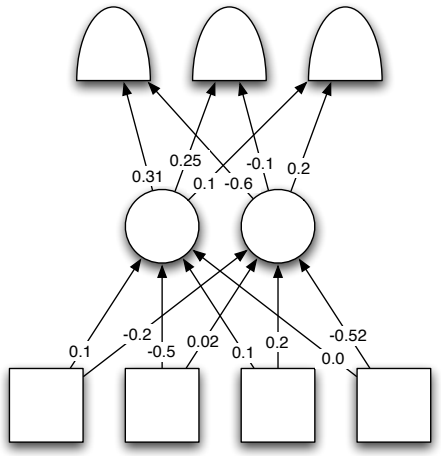

Figure 26 A multilayer perceptron with no bias units.

will be known as $\vec{o}_i$. Likewise the correct vector will be $\vec{y}$ with each element as $\vec{y}_i$. Our new hidden layer will be designated with the column vector H, and each element will be known as $\vec{h}_j$. Since j is taken by the hidden layer, our input layer $\vec{x}$ will have its elements now known as $\vec{x}_k$.

Because we have two layers of synapses, we now have two matrices which define their weights: W and V. The connection to an output neuron $\vec{o}_i$ from a hidden neuron $\vec{h}_j$ has a weight W_{ij}. The connection to a hidden neuron $\vec{h}_j$ from an input neuron $\vec{x}_k$ has a weight V_{jk}.

Our forward functions now look like this:

$$\begin{aligned}\vec{o}_i &= \sigma(\sum_j W_{ij}\vec{h}_j) \\ \vec{h}_j &= \sigma(\sum_k V_{jk}\vec{x}_k)\end{aligned}$$

Or removing the middle man...

$$\vec{o}_i = \sigma(\sum_j W_{ij}\sigma(\sum_k V_{jk}\vec{x}_k))$$

Or in matrix form,

$$\begin{aligned}\vec{o} &= \sigma(W\vec{h}) \\ \vec{h} &= \sigma(V\vec{x})\end{aligned}$$

Or...

$$\vec{o} = \sigma(W\ \sigma(V\vec{x}))$$

The algorithm is exactly the same, but the backpropagation learning rule is different:

$$\begin{aligned}\Delta W_{ij} &= \alpha(\vec{y}_i - \vec{o}_i)\vec{o}_i(1-\vec{o}_i)\vec{h}_j \\ \Delta V_{jk} &= \alpha\sum_i(\vec{y}_i - \vec{o}_i)\vec{o}_i(1-\vec{o}_i)W_{ij}\vec{h}_j(1-\vec{h}_j)\vec{x}_k\end{aligned}$$

It's important to note that ΔV_{jk} depends on the value of W_{ij}. **Thus you must not change $\mathbf{W_{ij}}$ (by adding the new $\mathbf{\Delta W_{ij}}$ to it) until *after* you have computed $\mathbf{\Delta V_{jk}}$.**

Note that ΔV_{jk} repeats part of ΔW_{ij}, and so the above equation can be rewritten:

$$\begin{aligned}\Delta W_{ij} &= \alpha(\vec{y}_i - \vec{o}_i)\vec{o}_i(1-\vec{o}_i)\vec{h}_j \\ \Delta V_{jk} &= \sum_i W_{ij}\Delta W_{ij}(1-\vec{h}_j)\vec{x}_k\end{aligned}$$

You can't quite do it as prettily in matrix form. Here's the long way:

$$\Delta W = \alpha\ ((\vec{y} - \vec{o}) \times \vec{o} \times (1 - \vec{o}))\ \vec{h}^T$$
$$\Delta V = \alpha\ \left(\vec{h} \times (1 - \vec{h}) \times (W^T((\vec{y} - \vec{o}) \times \vec{o} \times (1 - \vec{o}))\right)\ \vec{x}^T$$

We can split this up a little bit into two temporary column vectors $\vec{d}$ and $\vec{e}$. We'll reuse $\vec{d}$:

$$\begin{aligned}
\vec{d} &= (\vec{y} - \vec{o}) \times \vec{o} \times (1 - \vec{o}) \\
\Delta W &= \alpha\ \vec{d}\ \vec{h}^T \\
\vec{e} &= \vec{h} \times (1 - \vec{h}) \times (W^T \vec{d}\ ) \\
\Delta V &= \alpha\ \vec{e}\ \vec{x}^T
\end{aligned}$$

Proof Let's derive this rule like we did before. We start with the first rule (ΔW_{ij}) which is practically identical to the proof before, except it uses $\vec{h}_j$ instead of $\vec{x}_j$:

$$\begin{aligned}
\Delta W_{ij} &= -\alpha \frac{\partial \mathcal{E}}{\partial W_{ij}} \\
&= -\alpha \frac{\partial \mathcal{E}}{\partial \vec{o}_i} \frac{\partial \vec{o}_i}{\partial W_{ij}} \\
&= -\alpha (\vec{y}_i - \vec{o}_i)(-1) \frac{\partial \vec{o}_i}{\partial W_{ij}} \\
&= -\alpha (\vec{y}_i - \vec{o}_i)(-1) \vec{o}_i (1 - \vec{o}_i) \frac{\partial \sum_j W_{ij} \vec{h}_j}{\partial W_{ij}} \\
&= \alpha (\vec{y}_i - \vec{o}_i) \vec{o}_i (1 - \vec{o}_i) \vec{h}_j
\end{aligned}$$

No sweat.

But the second rule (ΔV_{jk}) has a summation in it. It's a bit longer but not too bad. Note that we're going to go down the error slope due to the V_{jk} value, not the W_{ij} value:

$$\begin{aligned}
\Delta V_{jk} &= -\alpha \frac{\partial \mathcal{E}}{\partial V_{jk}} \\
&= -\alpha \frac{\partial \mathcal{E}}{\partial \vec{h}_j} \frac{\partial \vec{h}_j}{\partial V_{jk}}
\end{aligned}$$

Now we expand the first term a little bit more. Here the effect on $\mathcal{E}$ of changing $\vec{h}_j$ is due to its effect on *every single* $\vec{o}_i$, and consequently their sum total effect on $\mathcal{E}$. So we have:

$$\Delta V_{jk} = -\alpha \sum_i \frac{\partial \mathcal{E}}{\partial \vec{o}_i} \frac{\partial \vec{o}_i}{\partial \vec{h}_j} \frac{\partial \vec{h}_j}{\partial V_{jk}}$$

We've already figured out the first term: $\frac{\partial \mathcal{E}}{\partial \vec{o}_i} = (\vec{y}_i - \vec{o}_i)(-1)$. The second term $\frac{\partial \vec{o}_i}{\partial \vec{h}_j}$ is determined similarly to how we've done it in the past:

$$\begin{aligned}
\frac{\partial \vec{o}_i}{\partial \vec{h}_j} =& \sigma(\sum_j W_{ij}\vec{h}_j)(1 - \sigma(\sum_j W_{ij}\vec{h}_j))\frac{\partial(\sum_j W_{ij}\vec{h}_j)}{\partial \vec{h}_j} \\
=& \vec{o}_i(1 - \vec{o}_i)\frac{\partial(\sum_j W_{ij}\vec{h}_j)}{\partial \vec{h}_j} \\
=& \vec{o}_i(1 - \vec{o}_i)W_{ij}
\end{aligned}$$

Likewise the third term $\frac{\partial \vec{h}_j}{\partial V_{jk}}$ is:

$$\begin{aligned}
\frac{\partial \vec{h}_j}{\partial V_{jk}} =& \sigma(\sum_k V_{jk}\vec{x}_k)(1 - \sigma(\sum_k V_{jk}\vec{x}_k))\frac{\partial(\sum_k V_{jk}\vec{x}_k)}{\partial V_{jk}} \\
=& \vec{h}_j(1 - \vec{h}_j)\frac{\partial(\sum_k V_{jk}\vec{x}_k)}{\partial V_{jk}} \\
=& \vec{h}_j(1 - \vec{h}_j)\vec{x}_k
\end{aligned}$$

And so the result is

$$\begin{aligned}
\Delta V_{jk} &= -\alpha \sum_i \frac{\partial \mathcal{E}}{\partial \vec{o}_i}\frac{\partial \vec{o}_i}{\partial \vec{h}_j}\frac{\partial \vec{h}_j}{\partial V_{jk}} \\
&= -\alpha \sum_i (\vec{y}_i - \vec{o}_i)(-1)\vec{o}_i(1 - \vec{o}_i)W_{ij}\vec{h}_j(1 - \vec{h}_j)\vec{x}_k \\
&= \alpha \sum_i (\vec{y}_i - \vec{o}_i)\vec{o}_i(1 - \vec{o}_i)W_{ij}\vec{h}_j(1 - \vec{h}_j)\vec{x}_k
\end{aligned}$$

4.5 The Error Backpropagation Algorithm

Given the above, the error backpropagation algorithm is relatively straightforward:

Algorithm 10 *Build a Two-Layer Neural Network with Error Backpropagation (Sigmoid)*

$X \leftarrow \{\langle \vec{x}^{(1)}, \vec{y}^{(1)} \rangle, ..., \langle \vec{x}^{(n)}, \vec{y}^{(n)} \rangle\}$ samples ▷ $\vec{x}^{(i)}$ and $\vec{y}^{(i)}$ are column vectors
$m > 0 \leftarrow$ desired number of hidden units
$\alpha > 0 \leftarrow$ learning rate ▷ Use a small value
$d \leftarrow$ stopping criterion

$V \leftarrow$ a matrix with $|\vec{x}^{(1)}|$ columns and m rows initially of small random values centered at 0
$W \leftarrow$ a matrix with m columns and $|\vec{y}^{(1)}|$ rows, initially of small random values centered at 0
$\Delta V \leftarrow$ a matrix identical to V
$\Delta W \leftarrow$ a matrix identical to W
$\vec{o}, \vec{p} \leftarrow$ vectors the same size as $\vec{y}^{(1)}$, initially zero ▷ We'll use these to determine when to stop
$\vec{h} \leftarrow$ vector of size m ▷ Hidden units
repeat
    $\vec{p} \leftarrow \vec{o}$
    Shuffle the samples in X randomly
    for i from 1 to n **do**
        $\vec{h} \leftarrow \sigma(V\vec{x}^{(i)})$
        $\vec{o} \leftarrow \sigma(W\vec{h})$ ▷ Use Algorithm 11
        $\Delta W \leftarrow \alpha\ ((\vec{y} - \vec{o}) \times \vec{o} \times (1 - \vec{o}))\ \vec{h}^T$
        $\Delta V \leftarrow \alpha\ \left(\vec{h} \times (1 - \vec{h}) \times (W^T((\vec{y} - \vec{o}) \times \vec{o} \times (1 - \vec{o}))\right)\ (\vec{x}^{(i)})^T$
        $W \leftarrow W + \Delta W$
        $V \leftarrow V + \Delta V$
until for each $x^{(i)}, |p^{(i)} - o^{(i)}| < d$ ▷ Things aren't changing much more
return V and W

Some notes. First, note that we've added a **stopping criterion** d to the algorithm. This simply means that when the outputs of the neural network aren't changing much more (the difference between the previous outputs and the current outputs is no more than d), we stop the algorithm. We do *not* stop the algorithm when the error drops below some value: because it may never do that if it gets caught in a local optimum.

Second, note that we've seeded V and W with small random values centered around zero. There's a bit of an art to picking these values: if you pick big values or bias the values in some way you may be able to converge faster, or may get caught in a local optimum with more likelihood. I'd go with small numbers (like 0.01 if your inputs are around 1.0).

Third, note that we're shuffling the samples and then iterating through them rather than picking samples from X at random with replacement.

Once you have a neural network, querying it is pretty straightforward too: it's just matrix multiplies and sigmoid:

Algorithm 11 *Query a Two-Layer Neural Network (Sigmoid)*

$\vec{x} \leftarrow$ query input ▷ $\vec{x}$ is a column vector
$V, W \leftarrow$ neural network matrices

$\vec{h} \leftarrow \sigma(V\vec{x})$
$\vec{o} \leftarrow \sigma(W\vec{h})$
return $\vec{o}$

Since backpropagation is a form of optimization, you need to know how well you've approached the minimum error (0). Here's the basic idea, using our sum-squared error metric discussed earlier:

Algorithm 12 *Compute the Error of a Two-Layer Neural Network (Sigmoid)*

$X \leftarrow \{\langle \vec{x}^{(1)}, \vec{y}^{(1)} \rangle, ..., \langle \vec{x}^{(n)}, \vec{y}^{(n)} \rangle\}$ samples ▷ $\vec{x}^{(i)}$ and $\vec{y}^{(i)}$ are column vectors
$V, W \leftarrow$ neural network matrices

$\mathcal{E} \leftarrow 0$
for i from 1 to n **do**
    $\vec{h} \leftarrow \sigma(V\vec{x}^{(i)})$
    $\vec{o} \leftarrow \sigma(W\vec{h})$
    $\mathcal{E} \leftarrow \mathcal{E} + (\vec{o} - \vec{y}^{(i)})^T(\vec{o} - \vec{y}^{(i)})$ ▷ Fancy way of doing sum-squared error
return $\mathcal{E}$

It's entirely possible that your network will simply never converge to anything remotely like a zero error: some sample is permanently stuck in a local optimum. You might modify this algorithm to output errors for each of the $\vec{y}^{(i)}$ in addition to the sum-squared error, so you can see which samples are having difficulty.

4.6 Hebb's Rule and Hopfield Networks

The Hopfield model is a neural network which simulates an important feature of learning: associative memory. An associative memory associates a key with a value. In the computer world, the most common associative memory one sees is a hash table. But hash tables have a weakness: they can only look up items in the table for wh ich a key exists.

What if you don't know the key exactly, but you know what it is approximately? In a hash table, this doesn't help you. But in a Hopfield network, you can still find the item. In a Hopfield network, the key and the value are the same thing — that is, the item is hashed with itself as key. When you go to a Hopfield network, you're asking it to find the item in its memory which most closely resembles the key you're providing.

Associative memory is how we store information in our brains. We have no idea how it works exactly, but there's a fair chance that it resembles the mechanisms inside the Hopfield network. One reason for this is that the Hopfield network uses a learning rule based on a real cognitive learning rule called **Hebb's Rule.**

Donald Hebb (1904–1985) discovered a mechanism of learning in synapses that went like this. If neuron A is firing pulses to neuron B, and neuron B fires when neuron A fires, then the strength

of the synapse between A and B increases. That is, if there is a positive correlation between A's pulses and B's pulses, then the synapse strength grows. Hebb said nothing about what would happen when there was a negative correlation; but the Hopfield model, and other Hebbian models, use an"extended" version of the Hebb rule that says that when there is a negative correlation, the synaptic strength decreases. If you have neuron x_j feeding into o_i, then this generalized Hebbian rule is:

$$\Delta w_{ij} = \alpha o_i x_j$$

Again, α is the learning rate. Contrast this with the "delta rule" used in backpropagation. Here's the simplest form (you saw this before, as Equation 4.2 in the Perceptron section, on page 55):

$$\Delta w_{ij} = \alpha(y_i - o_i)x_j$$

Note that $y_i - o_i$ is the *error* of o_i. So whereas Hebb's rule changes the weights proportional to the correlation between the input node and the output node, the Delta rule changes the weights proportional to the correlation between the input node and the *error* in the output node.

Unlike the delta rule, which works well for any set of inputs, Hebb's rule only works really well if the various things being learned are orthogonal to each other. Otherwise you start getting some crosstalk leakage between learned items. So it's a much weaker rule. Also unlike the delta rule, Hebb's rule exists in real biology.

The Hopfield network uses a form of the generalized Hebb rule to store boolean vectors in its memory. When the user presents the network with an input, the network will produce the item in its memory which most "closely resembles" that input. Unlike in backpropagation, the Hopfield network is a fully-connected, recurrent neural network. That is, every neuron is connected to every other neuron. Figure 27 shows a small Hopfield network.

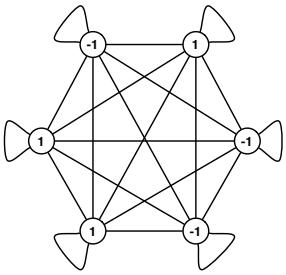

Figure 27 A Hopfield Network with neuron values.

Notice that the six neurons are all connected to each other — they're even connected to themselves. The neurons can only have the values 1 or -1. Also notice that the edges are undirected. This Hopfield network has six neurons, which means that it can learn vectors of size six. Let's say we want this network to learn the following pattern: 010011. Hopfield networks learn binary patterns, but not 1's and 0's. Instead, they learn 1's and -1's. So our converted pattern becomes: (-1)1(-1)(-1)11. To teach the network this pattern, we first assign a neuron to each element in the pattern, as we had done in Figure 27, going clockwise from top left. Next, we assign the weights of the edges, according to the following rule:

$$W_{ij} = \frac{1}{N} x_i x_j$$

N is the number of neurons in the network ($N = 6$ in our network). Notice how similar this rule is to the generalized Hebbian learning rule. Thus our edges look like Figure 28:

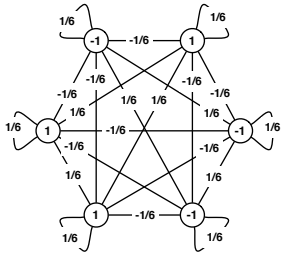

Figure 28 Hopfield Network Edge Weights.

To get the Hopfield network to find the "closest" image in its memory to our key value, we first set the neurons in the network to the key value (our input value). Then we follow the following procedure:

1. Pick some neuron i at random.
2. Update X_i according to the following rule:

$$x_i = \text{sign}\left(\sum_j W_{ij} x_j\right)$$

$$\text{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{if } u < 0 \end{cases}$$

3. Go to #1 until no neurons in the network are changing any more (the network has converged to an answer).

When the neural network has converged, the values of its neurons are the output (the memory item closest to our input).

Presently our network only has one thing in its memory, so it will always converge to that. That's not too interesting. What we want is to store multiple items in its memory. To do this, we set the weights to the sum of the correlations for all the items we want to store. Let P be the set of input vectors we're trying to learn. Then learning rule for the weights now looks like this:

$$W_{ij} = \frac{1}{N} \sum_{p \in P} x_i^{(p)} x_j^{(p)}$$

Now when you present an input to the network, it will converge to whichever of these memorized vectors is closest. The Hopfield network with N neurons can reliably store about $0.138N$ vector patterns in its memory. This is called the *capacity* of the network. If you store more than this, then the network begins to degrade. What's interesting is that the degradation is gradual, and very similar to how our brains degrade when we're jamming too much information to learn at one time. The Hopfield network begins to lump similar learned patterns together into a kind of mixture of the two, that's part one pattern and part the other.

Hopfield is basically a classifier which does a kind of nearest-neighbor algorithm.

5 Competitive Networks, Self-Organizing Maps, and Kohonen's Algorithm

So far we've seen supervised methods. Next we'll look at some unsupervised uses of neural networks. One approach is known as *competitive learning*. In competitive learning, neurons fight amongst themselves for who gets to say what category a particular input belongs in. The neurons "closest" to the input are the strongest, and they will ultimately win the contest. Here's a simple competitive learning network architecture:

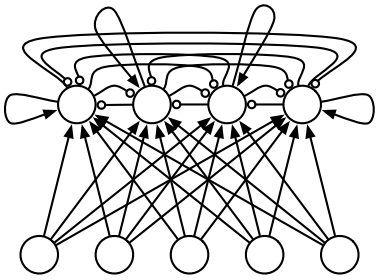

Figure 29 A Competitive Architecture.

Doesn't look simple, does it? Well, it's actually pretty simple. The input neurons on the bottom all feed into the output neurons on the top. Additionally, the output neurons are self-connected

with positive weights. Lastly, the each output neuron has an inhibitory (negative-weight) connection to all other output neurons (that's what the edges with circles at their ends are) The inputs are real-valued.

The network shown will cluster the 5-value vectors into 4 categories (one category per output neuron) as follows. We first present a vector to the input. The output neurons' outputs are then computed as simply the sum of their inputs (weighted by the edge weights). We then recompute the outputs again: since output neurons' outputs are then fed as input into other output neurons, the values will be different. We repeat the re-computation over and over. Eventually one neuron will come to dominate, and the outputs of the other neurons will get smaller, because as it gets bigger, the dominating neuron's values, pushed through the inhibitory connections, will affect the other neurons negatively. Once a neuron is dominating, we designate it as the **winner.**

The learning rule for this network only changes the weights going from the input neurons to the winner output neuron. Let us designate this neuron i^*. We'll adjust the weights so that this input makes the winner neuron even stronger than it was. This is done with the following equation (we assume that $\vec{x}$ is the input layer and $\vec{y}$ is the output layer):

$$\Delta W_{i^*j} = \alpha(x_j - W_{i^*j})$$

... that is, the weights are moved a little away from their previous settings and towards the input neuron's values. The initial values of the weights from the input neurons should, as in backprogation, be small random values. The trick to getting this neural network working is figuring out what the values of of the weights should be from other output neurons (and the self-weight). These are often set to some small value, say, 0.1, and are never changed. But it's up to the network designer.

Sometimes one output neuron will never gain enough strength to win anything, and thus our network will cluster into 3 categories and not 4. Such a neuron is known as a **dead unit**. There are a variety of ways of dealing with dead units, but one popular approach is called leaky learning. In **leaky learning**, not only are the winner's weights updated, but so are the loser's weights — only to a much smaller degree. This allows a constantly-losing neuron to gradually get stronger and stronger until it can finally compete with the others.

This notion of having inhibitory weights, causing output neurons to fight for supremacy, is known as **lateral inhibition.** Lateral inhibition is a particularly popular notion in neural networks because it is a concept borrowed directly from neuroscience — the cerebral cortex does lateral inhibition all the time, and it appears to play a major role in how we are able to cluster patterns together in order to do pattern recognition.

One particular kind of lateral inhibition is of special interest: a kind of lateral inhibition which only applies to the nearest neighbors of the output neuron. This kind of inhibition results in a dynamic known as **feature mapping.** Imagine a neural architecture where you have inputs feeding to outputs, but the outputs are arranged locally, say, in a 2-d grid. Each output neuron has a positive weighted connection to its nearby neighbors and an inhibitory connection to neighbors slightly further away. No connections are made to far-away neurons. The diagram at right shows this; however, only one output neuron's connections are shown, to make the point clearer, so you'll have to imagine all of the neurons wired up with edges to each other.

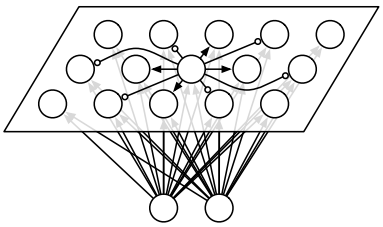

Figure 30 2-D Lateral Inhibition.

You could do this in 1 or 3 dimensions also (any number of dimensions in fact). Here's a similar arrangement in 1 dimension — once again, only one output neuron's outgoing edges are shown for brevity. You can have as many inputs as you like, btw — I'm just showing two inputs here.

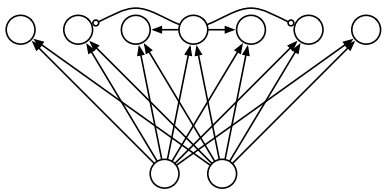

Figure 31 1-D Lateral Inhibition.

This is a very biological way of hooking things up — it's done very similarly to this in the cerebral cortex. But it doesn't have to be this way — all of the connections an output neuron might be making could be positive, just that they gradually get weaker. Or they might be very strong for nearby neurons, less strong for further out neurons, negative for even further neurons, and then disconnected (0) for way out neurons. The general notion is to come up with an architecture where an output neuron will help its nearby friends more than neurons further away.

What happens in competitive networks arranged like this? Well, now a neuron isn't all on its own as was the case in the earlier version of competitive networks; now the neuron has the help of its spatially local buddies. This results in an interesting phenomenon: the networks not only learn to cluster inputs into categories, but categories that are spatially related to each other are clustered by neurons closer to one another.

This also happens in the brain. The neurons of the sensory cortex, a chunk of the cerebral cortex responsible for touch sensations, are ordered in an interesting way. If two points on your body are close to each other, then the neurons responsible for those points are also close together. This results in the cortex being a kind of two-dimensional map of the body — at the top is the head, then the chest and arms, then the torso, then the legs and feet at the bottom. And certain areas which have much more touch sensitivity (the tongue, lips, hands, particularly the thumb, etc.) are spread out among many more neurons in the cortex than areas with relatively little touch sensitivity (the stomach, say). The neurons have learned how to spatially divvy up the body's touch inputs in as evenhanded a way as possible. This spatial divvying-up goes by a special name: a self-organizing map.

We'll finish up with Kohonen's algorithm, a neural network which does a shortcut to the competitive mechanisms discussed earlier. Kohonen's algorithm does a remarkable job of spatially divvying up the input data into clusters in the output, as I had shown in pictures in class.

Kohonen's network dispenses with the lateral inhibition stuff, so it's very simple. Here are 1-D and 2-D versions of the network:

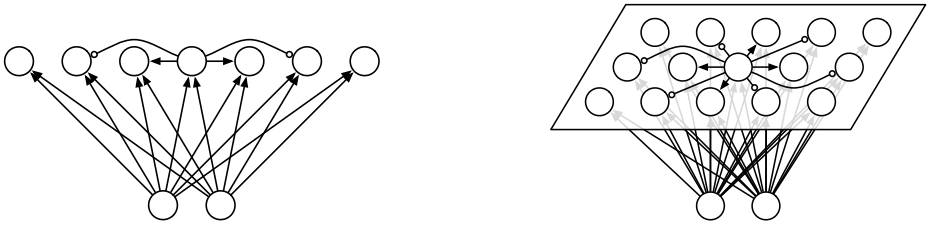

Figure 32 1-D and 2-D Kohonen Architectures (Compare to Lateral Inhibition, Figures 31 and 30.

Notice there are no lateral connections. The initial weights are small random values as usual. To train a Kohonen net, you first present the input to the input neurons. The output neurons' values are simply the sum of their inputs (multiplied by the weights as usual). Then you pick the "winner"

output neuron i^*, which is just the neuron with the largest output value. All edge weights are then updated according to the following function (where $\vec{x}$ is the input layer and $\vec{y}$ is the output layer, as usual):

$$\Delta W_{ij} = \alpha \Lambda(i, i^*)(x_i - W_{ij})$$
$$\Lambda(i, i^*) = e^{\frac{-|\vec{r}_i - \vec{r}_{i^*}|^2}{2\beta^2}}$$

α is the learning rate as usual. The $\Lambda(i, i^*)$ function is a neighborhood function on a neuron i with respect to the winner i^*. Specifically, $\vec{r}_i$ and $\vec{r}_{i^*}$ are the positions of neuron i and i^* respectively **embedded in the plane (or line) of output neurons.** Thus the lambda function is closer to 1 when i and i^* are very near neighbors, but is closer to 0 when they are far away. This function is basically similar to a bell curve centered at i^*. The parameter β controls the variance of the curve, and it is commonly decreased gradually as the network is trained.

So what this function is doing is modifying the weights of output neurons based on their proximity to the winner. Neurons close to the winner get their weights modified greatly in the direction of the input neurons' values. Neurons far from the winner get their weights modified very little. This is effectively a shortcut to all the lateral inhibition stuff, and it does a very good job.