Supervised learning in single-stage feedforward networks

Bruno A. Olshausen

September 1, 2014

Abstract

This handout describes supervised learning in single-stage feedforward networks composed of McCulloch-Pitts type neurons. These methods were originally worked out by Bernie Widrow at Stanford in the early 1960's. Related work was done by Frank Rosenblatt at Cornell around the same time. There are numerous texts that describe these ideas, among them the text by Herz, Krogh, & Palmer, as well as Widrow's book, "Adaptive Signal Processing" (Widrow & Stearns, Prentice-Hall, 1985). The idea of this handout is to describe the central ideas in a rather condensed and intuitive form.

The problem we consider is that of learning in single-stage networks like that shown in Figure 1. Information flows strictly upward in this network, from the inputs

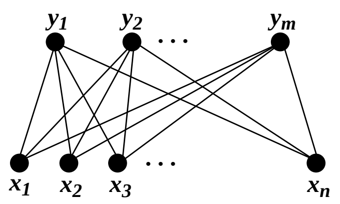

Figure 1: A single-stage network of model neurons.

x_i at the bottom to the outputs y_i at the top. Thus, it is termed a "feedforward" network. For example, the input nodes may correspond to some sensory input and the output nodes may represent some appropriate motor response for a simple organism. The function performed by the network is essentially determined by the links connecting the input nodes to the output nodes, analogous to synapses in real neurons. Our challenge is to figure out how to change these links so as to achieve some desired input-output transformation. Importantly, we would like the system to *learn* the proper links from example, by training it with desired input-output pairs. This form of learning is called *supervised learning* because it relies on having a "teacher" to tell the system what the desired output is for each input example. In order to make

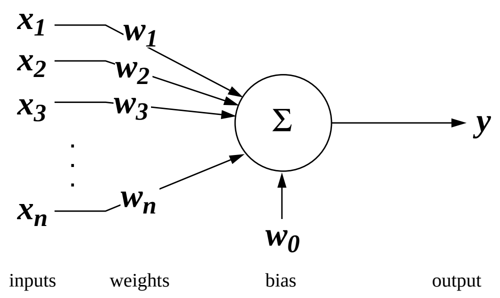

Figure 2: A model linear neuron.

this work, we need to devise a *learning rule* for the system, which specifies how the links are automatically changed upon each trial of training. We shall first consider the problem of learning in a single linear neuron, and then we will add a non-linearity on the outputs. We then discuss what sorts of transformations are possible to learn with such a system.

A single linear neuron

Let us first consider how to learn with a single, linear neuron, illustrated in Figure 2. The inputs are denoted by the variables, $x_1, x_2, ..., x_n$; the links between the input and output, or *weights*, are denoted by $w_1, w_2, ..., w_n$; and the "bias" to the neuron is denoted by w_0. The output, y, is a linear function of the inputs, x_i, as follows:

$$y = \sum_{i=1}^{n} w_i \, x_i + w_0 \, . \tag{1}$$

The input-output transformation, or the function performed by the neuron depends entirely on the weights, w_i, in addition to the bias term w_0. It will generally be assumed that these parameters are initially set to random values. The goal of learning is to tweek the weights and bias over time so that a particular, desired input-output function is performed by the neuron.

For example, let us say that we wish to train the system to perform a pattern classification task, so that it yields a $+1$ response to a 'T' pattern presented on the inputs, and a -1 response to an 'L' pattern presented on the inputs, as shown in Figure 3*a*. With a little effort one can see that the weight values that would achieve this are those shown at right, in Figure 3*b*, with the bias w_0 set to zero. (Verify for yourself that this is correct.) But we could see the solution here only because we are smart. If you were a little neuron, and all you received were a bunch of inputs and their desired corresponding outputs as training examples, what rule would you use to adjust your weights to achieve the desired outcome?

Here's an intuitive way to think about the problem: Let's say you are presented with a specific pattern on your inputs, you compute the corresponding output, y,

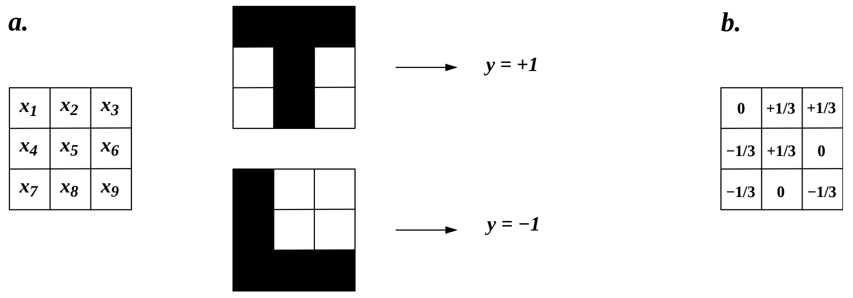

Figure 3: A pattern discrimination task. *a*, The inputs are arrayed in two-dimensions as shown at left. We wish to train the neuron to yield a +1 response to a 'T' and a -1 response to an 'L.' Black pixels denote $x_i = 1$, blank pixels denote $x_i = 0$. *b*, The weight values, w_i, that achieve the desired result, displayed with the same ordering as the x_i.

according to Equation 1, and it does not match the desired output. The first question you might ask is whether the computed output, y, was too high or too low. If y was too low, then if x_i is "on" (i.e., a black pixel) we should crank up w_i on that trial. That way, the next time the pattern is presented, y will be closer to the desired value. Vice versa, if y is too high, then we should decrease w_i.

Let's see if we can put these intuitions on a firmer foundation. We will enumerate the training examples with the symbol α, where each "example" consists of an input pattern along with its desired output. In the case above, we would have $\alpha = 1, 2$ for the training examples 'T'/+1 and 'L'/-1 respectively. We shall denote the input state for example α as $\mathbf{x}^\alpha = \{x_1^\alpha, ...x_n^\alpha\}$, and the desired output (or teacher signal) for example α as t^α. (Note that the α are merely superscripts which denote a specific training example, not an exponent.) The goal of learning is to find a set of w's that minimize the error between the desired output or teacher signal, t^α, and the actual output computed via Equation 1, $y(\mathbf{x}^\alpha)$, over all of the examples α. If we take the measure of the error to be the square of the difference between the teacher and computed output, then our error function for example α is

$$E^\alpha = [t^\alpha - y(\mathbf{x}^\alpha)]^2 \tag{2}$$

and the total error function over all the examples is then

$$E = \sum_\alpha E^\alpha \tag{3}$$

assuming each training example is weighed equally. E is essentially a function of the weights and bias of the neuron, $w_0 ... w_n$. It turns out that since the neuron is linear in this case, we can compute the solution for the w's in closed-form (we will show this in class). But for now we are not going to assume that our little neuron is this smart, and so we need to come up with an iterative method for determining the w's, assuming that we have a teacher who is patient enough to make repeated

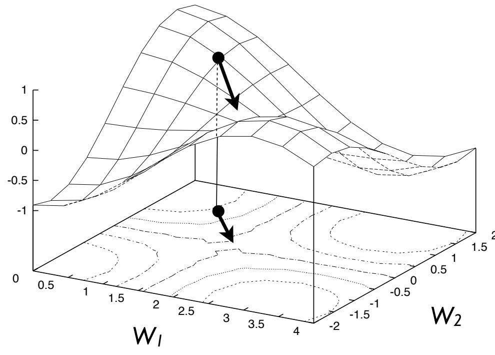

Figure 4: A function of two variables, $f(w_1, w_2)$, which describes the surface shown. The gradient evaluated at any point tells you which way is downhill at that point.

presentations of input-output pairs until the neuron finally gets it right. In order to do this, we need to find out how to tweek the w's on each training example so as to reduce E^α. For this, we shall turn to a technique known as *gradient descent.*

An aside: gradient descent

Gradient descent is a technique used for function minimization. It is based on a very simple principle: *move downhill.* Let's say we have a function of two variables, $f(w_1, w_2)$, as illustrated in Figure 4. The question is, if we are currently sitting at the point (w_1^0, w_2^0), in what direction should we move in order to go downhill? If you were a skier, you would simply inspect the slope where you are standing and point the skiis downward. If you are a variable, you do the same thing. The only problem is that you need to figure out which way is down (because variables aren't equipped with visual perception). If you have a formula for the function, f, which describes the surface, then the gradient of f tells you which way to go to increase the function, and so the negative gradient tells you which way is down. The slope along the w_1 direction is $\frac{\partial f}{\partial w_1}$, while the slope along the w_2 direction is $\frac{\partial f}{\partial w_2}$. So, if you are standing at (w_1^0, w_2^0), you simply evaluate the slopes in each direction at that point and move in a direction proportional to the negative gradient:

$$\Delta w_1 \propto -\frac{\partial f}{\partial w_1}\bigg|_{w_1^0, w_2^0} \tag{4}$$

$$\Delta w_2 \propto -\frac{\partial f}{\partial w_2}\bigg|_{w_1^0, w_2^0} \tag{5}$$

where $|_{w_1^0, w_2^0}$ means "evaluated at w_1^0, w_2^0." The minus sign indicates that we are moving downhill. The exact amount you move depends on how bold you are, as well as the smoothness of the surface. The derivative is only a *local* measurement of the slope, so as you move away from (w_1^0, w_2^0) the slope will generally change. If you step too far, you will end up bouncing all around the surface rather than going downhill like you intended. Picking the right step size is one of the tricky empirical aspects of gradient descent. Usually, you will want to observe how f decreases as you increase your step size, and pick a step size that yields the greatest average decrease in f per step.

OK, now back to the problem of determining the learning rule for our linear neuron. E^α is a function of the w's, just like f above, so we can get a learning rule for the weights and bias simply by doing gradient descent:

$$\Delta w_i \propto -\frac{\partial E^\alpha}{\partial w_i}\,, \tag{6}$$

where Δw_i denotes the incremental change we should make in weight w_i on trial α. Now we just have to do some math to figure out what $\frac{\partial E^\alpha}{\partial w_i}$ is. Using the definition of E^α in Equation 2 we get

$$\begin{aligned} \frac{\partial E^\alpha}{\partial w_i} &= \frac{\partial}{\partial w_i}\left[t^\alpha - y(\mathbf{x}^\alpha)\right]^2 && (7)\\ &= -2\left[t^\alpha - y(\mathbf{x}^\alpha)\right]\frac{\partial}{\partial w_i} y(\mathbf{x}^\alpha)\,. && (8) \end{aligned}$$

Then, using the definition of $y(\mathbf{x}^\alpha)$ in Equation 1 we get

$$\begin{aligned} \frac{\partial}{\partial w_i} y(\mathbf{x}^\alpha) &= \frac{\partial}{\partial w_i}\sum_j w_j x_j^\alpha + w_0 && (9)\\ &= \begin{cases} x_i^\alpha & \text{if } i \geq 1 \\ 1 & \text{if } i = 0 \end{cases}\,. && (10) \end{aligned}$$

Thus, our learning rule is

$$\begin{aligned} \Delta w_i &\propto \left[t^\alpha - y(\mathbf{x}^\alpha)\right] x_i^\alpha && (11)\\ \Delta w_0 &\propto \left[t^\alpha - y(\mathbf{x}^\alpha)\right]\,. && (12) \end{aligned}$$

Equation 11 says that on each trial, α, we change w_i by an amount proportional to the output error times the current input, x_i^α. For example, if the i^{th} input on example α is "on" ($x_i^\alpha = 1$), and the computed output, $y(\mathbf{x}^\alpha)$, is lower than the desired output t^α, then w_i should be increased. This is just as we intuited previously. Moreover, if we are off by alot, then w_i should change alot. Equation 12 says that the bias, w_0, increases or decreases proportional to the amount of undershoot or overshoot, respectively, in $y(\mathbf{x}^\alpha)$, independent of any of the input values.

A linear neuron with an output non-linearity

Now let us make a slight modification to the neuron so that there is a non-linearity on the output, as shown in Figure 5. The input-output relationship for this neuron

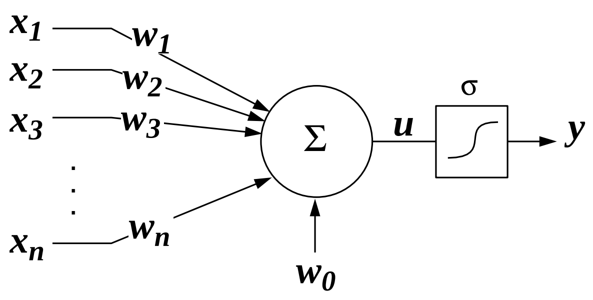

Figure 5: A linear neuron with an output non-linearity.

is then

$$
\begin{aligned}
y &= \sigma(u) && (13)\\
u &= \sum_{i=1}^{n} w_i\, x_i + w_0\,. && (14)
\end{aligned}
$$

The function σ expresses the non-linearity on the output, which we take to be the so-called *sigmoid function*:

$$\sigma(x) = \frac{1}{1 + e^{-\lambda x}}\,. \tag{15}$$

This function is plotted in Figure 6 for several values of λ. The effect of the output

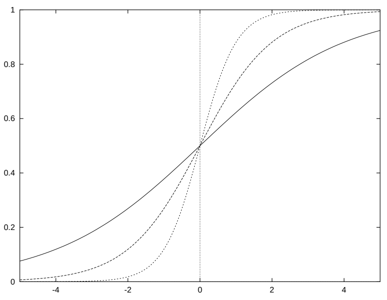

Figure 6: The sigmoid function $\sigma(x) = \frac{1}{1+e^{-\lambda x}}$. The function is plotted for several different values of λ which controls the steepness.

non-linearity is that it will cause y to “saturate” as it gets too large, and it will “bottom-out” as it gets too small. It between, it has a rather linear relationship to the sum over the inputs, u. Frank Rosenblatt considered the case of a neuron with a very sharp output linearity ($\lambda = \infty$), so that it essentially becomes a step function. In this case, you cannot derive a learning rule via gradient descent, because you cannot compute a gradient of the output with respect to the input (since the step function is discontinous). Rosenblatt instead devised an algorithm, presumably following his intuitions, and proved that it would converge on a solution (see Herz, Krogh and

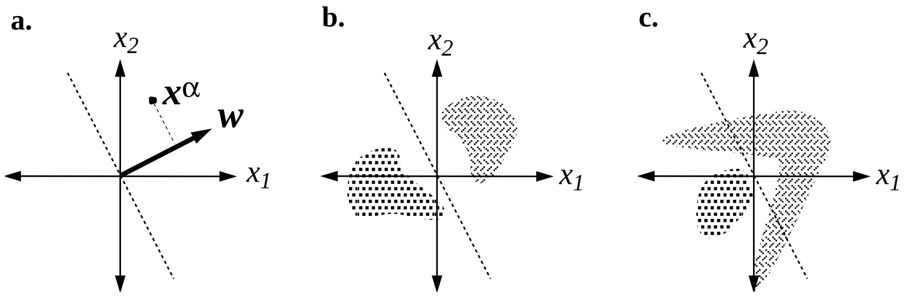

Figure 7: A geometric interpretation of the function performed by a model neuron. *a*, A linear neuron computes its output by simply taking the projection of a given input pattern $\mathbf{x}^\alpha$ onto the weight vector, and then adding bias w_0 (which is for now set to zero). The dotted-line through the origin denotes the separating line (or hyperplane, in higher dimensions) for separating positive outputs from negative outputs. *b*, Two pattern classes that may be separated by a line or plane are said to be *linearly separable*, and any such classes of patterns may be successfully discriminated by the model neuron. *c*, An example of two pattern classes that are not linearly separable, and which therefore could not be discriminated by a model neuron, no matter how hard it tried.

Palmer, chapter 5). But if the output non-linearity is somewhat gradual ($\lambda \approx 1$), then you can compute a gradient and derive a learning rule for the w_i. This you will do in the homework.

Linear separability

Can a single neuron, with or without an output non-linearity, discriminate any arbitrary sets of patterns? For example, let's say we want to make the system give a $+1$ response to either a 'T' or a 'U', and a -1 response to either an 'L' or an 'O'. Will this be possible? Let's make a geometric picture of what the model neuron is doing, and then we will see what its limitations are. Figure 7 shows the input state-space for a neuron with two inputs x_1 and x_2. A given input pattern, $\mathbf{x}^\alpha = \{x_1^\alpha, x_2^\alpha\}$ constitutes a point in this space. We can consider the weights to form a vector $\mathbf{w}$ in this space with components w_1, w_2. The output of the linear neuron is then given by projecting point $\mathbf{x}^\alpha$ onto the weight vector, $\mathbf{w}$ as shown in Figure 7*a*. This can be seen by re-writing Equation 1 in vector terms,

$$y = \mathbf{w}^T \cdot \mathbf{x} \qquad (16)$$

(ignoring for now the term w_0). Any point $\mathbf{x}$ lying along the dotted line, perpendicular to $\mathbf{w}$, will yield an output of zero. Any point lying to the right of the dotted line will yield a postive output, and any point lying to the left of the dotted line will yield a negative output. If we pass the output of the neuron through the logistic function non-linearity, using a high value of λ so as to make a steep transition, then those input patterns falling to the right of the dotted line will yield an output of 1 and those falling to the left will yield an output of 0. Thus, the class of patterns that can be discriminated by such a model neuron are those that can be separated by a line in two-dimensions, or by a *hyperplane* in higher dimensional spaces, as shown in Figure 7b. This is known as *linear separability*, and it is an important limitation to the discrimination capabilities of linear neuron models (with or without an output non-linearity). Pattern classes that are not linearly separable, such as in Figure 7c, cannot be discriminated by linear neurons.

Multiple outputs

It is now easy to extend the above neuron models to a system consisting of multiple neurons, each with a distinct output, as shown originally in Figure 1. We just need to re-denote the outputs, weights, and biases for each of the neurons as follows:

$$y_i = \sigma(u_i) \quad (17)$$

$$u_i = \sum_j w_{ij}\, x_j + w_{i0} \quad (18)$$

where w_{ij} denotes the weight from the j^{th} input node to the i^{th} output node. The learning rule for achieving a desired input-output transformation is the same as derived previously—it is just a matter of redoing the subscripts on the w's and y's.

The next challenge will be to have multiple layers of such neurons, so that one set of neurons receives its inputs from the outputs of a previous stage. This is how we will get around the limitation of linear separability. The problem then becomes how to learn the weights of the "hidden units" in such a multi-layer system.