Study Notes for AI (CS 440/520)
Lecture 22: Neural Networks

Corresponding Book Chapters: Chapter 18.7
Note: These notes provide only a short summary and some highlights of the material covered in the corresponding lecture based on notes collected from students. Make sure you check the corresponding chapters. Please report any mistakes in or any other issues with the notes to the instructor.

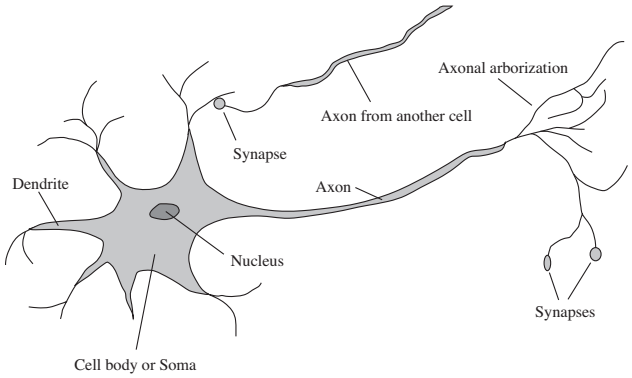

Figure 1: A schematic diagram of a single neuron

1 Background

A *neuron* is a brain cell whose main function is collecting, processing, and disseminating electrical signals (refer to Figure 1 above for a diagram of one of these cells). The brain's information-processing capacity is believed to be primarily due to the actions of networks of neurons. Because of this, there has been a large body of research devoted to recreating these networks of neurons artificially, and allowing computers to process information in the same way that human brains do. A simple model of one of the first neurons devised for computing can be seen in Figure 2. It works by 'firing' when a linear combination of its inputs exceeds some threshold. Neural networks can be very useful in their ability to perform distributed computation, to learn, and to handle noisy inputs. Even though other kinds of systems, such as Bayesian networks, have similar capabilities, neural networks are still one of he most popular and most effective types of system for machine learning.

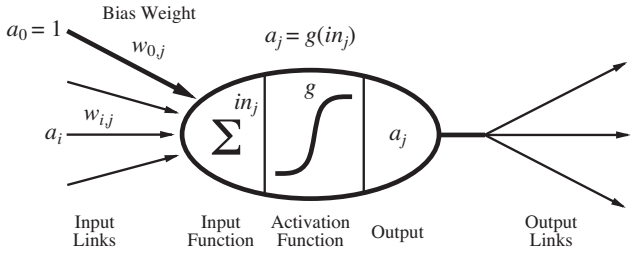

Figure 2: A model of a neuron. The unit's output is activated based on the weighted sum of all inputs a_j, with the weights for each a_j given as $W_{j,i}$ in the figure.

2 Units

Neural networks are compromised of units (see Figure 2) which are connected by directed links. A link from unit j to unit i will propagate the activation a_j from unit j to i. Each link is also assigned a weight $W_{j,i}$, which will determine both the strength of the connection, and the sign. Each unit will first compute a weighted sum of its inputs:

$$\sum_{j=0}^{n} W_{j,i} a_j \tag{1}$$

Then it will apply an activation function g to this sum, in order to derive the output (denoted as a_i):

$$a_i = g(\sum_{j=0}^{n} W_{j,i} a_j) \tag{2}$$

The activation function g is designed to meet two conditions. Firstly, we want the unit to be active (with output near $+1$) when the correct inputs are given to it, and inactive (with output close to 0) when the incorrect inputs are given. Secondly, the activation needs to be nonlinear, to keep the entire neural network from becoming a simple linear function. In Figure 3, we see two possible choices for the activation function g: the threshold function and the sigmoid function (or logistic function). The sigmoid function is useful because it is differentiable, which is important for the weight-learning algorithm that we will discuss later. Notice that in addition to the inputs and weights from the other nodes in the network, each node also gets a bias weight, $W_{0,i}$ attached to the fixed input $a_0 = -1$. This weight sets the actual threshold for the unit, in the sense that the unit is activated when the weighted sum of the other inputs exceeds $W_{0,i}$. This allows us to adjust the threshold of g away from 0, where it would normally be.

We can compare single units to basic logic gates in order to get a feel for how they operate. Figure 4 shows how the Boolean functions *and, or*, and *not* can be represented by threshold units with suitable weights. This is important because it means that using a network of units, we can compute *any* Boolean function of inputs.

3 Network Structures

There are two main types of neural networks: acyclic (feed-forward networks) and cyclic (recurrent networks). A feed-forward network represents a function of its current input, needing no internal state other than the weights themselves. A recurrent network, however, has to feed its outputs back into its own inputs. This means that the activation levels of the network will form a dynamic system that can exhibit numerous behaviors, including reaching a stable equilibrium or exhibiting oscillations or chaotic behavior. Additionally, recurrent networks can support

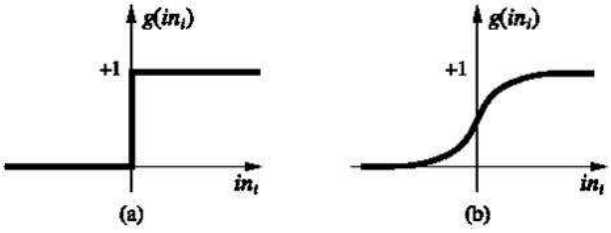

Figure 3: (a) The threshold activation function, which outputs 1 when the weighted sum of inputs is positive, and 0 otherwise. (b) The sigmoid function.

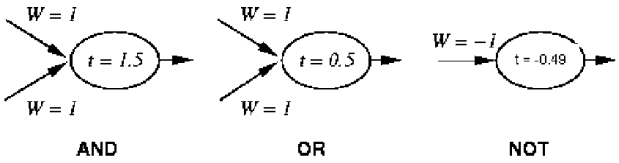

Figure 4: Units with a threshold function acting as logic gates, given appropriate input weights and bias weights

short-term memory, as the response of the network to a given input will depend on its initial state, which can in turn depend on previous inputs. This makes recurrent networks work better as models of the brain, but also makes them more difficult to understand.

Let us look more closely at the statement that a feed-forward network is merely a function of its inputs. Consider the network shown in Figure 5, which has two inputs, two hidden units, and an output unit. (We will eliminate the bias units in this example, for simplicity) Given an input $x = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$, and the network computes the output node:

$$a_5 = g(W_{3,5}a_3 + W_{4,5}a_4)$$
$$a_5 = g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2))$$

Thus, by expressing the output of each hidden unit as a function of *its* inputs, we can see that the output of the network as a whole can be written as a function of the inputs to the network. In addition, we can see that the weights in the network act as parameters for this function, so by changing the weights, we can change the function that the network itself represents. This is how learning occurs in neural networks.

3.1 Single layer feed-forward networks (perceptrons)

A network with all of the inputs connected directly to the outputs is called a single-layer network, or a perceptron network. Since each output unit is independent of the others, we can limit our study of perceptrons to networks with only a single output unit. First, let us look at the hypothesis space that a perceptron can represent. With a threshold activation function, we can view the perceptron as representing a Boolean function. In addition to the basic Boolean functions discussed before, a perceptron can also represent more complex functions, such as the majority function, which outputs a 1 only if more than half of its inputs are 1. This specific function can be represented by a perceptron with each weight W_j equal to 1 and the threshold weight equal to $n/2$.

Unfortunately, there are many Boolean functions that the threshold perceptron can not represent. Looking at equation 1, we can see that the threshold perceptron will only return 1 if the weighted sum of its inputs is positive,

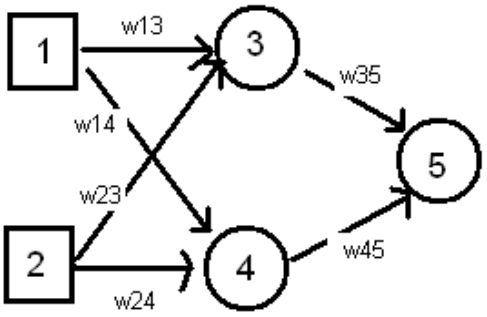

Figure 5: A simple neural network with two inputs, two hidden units, and an output

thus defining a hyperplane on the input space. Because of this, the threshold perceptron is called a linear separator. Figure 6 shows the hyperplane for the perceptron representations of the *and* and *or* functions of two inputs, where black dots represent a point where the value of the function is 1, and white dots indicate a point where it is 0. The perceptron can represent these because they are linearly separable, or there is some line that separates all the white dots from all of the black dots. However, we can also see in Figure 6 that the *xor* function is not linearly separable. Thus, there is no way for it (or any other non-linearly separable function) to be represented by a threshold perceptron.

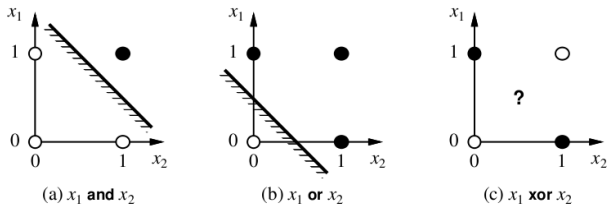

Figure 6: Illustration of linear separability in threshold perceptrons.

Despite the fact that they can only express a limited number of functions, threshold perceptrons do have an advantage in that there is a simple learning algorithm that will fit a threshold perceptron to any linearly separable training set. In addition, we can derive a similar algorithm for learning in sigmoid perceptrons.

The idea behind this algorithm is to adjust the weights of the network to minimize the error on the training set. Thus, learning is seen as an optimization search in the space of weights for the network. Traditionally, we view the error as the sum of squared errors, which, for a single training example with input x and true output y is written as:

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_w(x))^2 \tag{3}$$

where $h_w(x)$ is the output of the perceptron on the example, and y is the true output value. We can use gradient descent to reduce the squared error by calculating the partial derivatives of E with respect to each weight. We then have:

$$\frac{\delta E}{\delta W_j} = Err \times \frac{\delta Err}{\delta W_j} \tag{4}$$

$$= Err \times \frac{\delta}{\delta W_j} g(y - \sum_{j=0}^{n} W_j x_j) \tag{5}$$

$$= Err \times g'(in) \times x_j \tag{6}$$

where g' is the derivative of the activation function. We can now update the weight as follows:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j \tag{7}$$

Where α is the learning rate. Intuitively, this means that if the error is positive, then the network output is too small, and the weights need to be increased for the positive inputs, and decreased for the negative inputs. The opposite would be true if the error were negative.

3.2 Multi-layer feed-forward neural networks

Now let us consider networks with hidden units. In the most common case, we will have a single hidden layer (as in Figure 8). The advantage to be gained from adding hidden layers is that it will increase the size of the hypothesis space that the network can represent. If we think of each hidden unit as a perceptron that represents a soft threshold function in the input space, we can think of an output unit as a soft-thresholded linear combination of these functions. For example, if we add two opposite-facing soft threshold functions, and threshold the result, we can obtain a ridge function, and by combining two such ridges, we can get a bump (Figure 7).

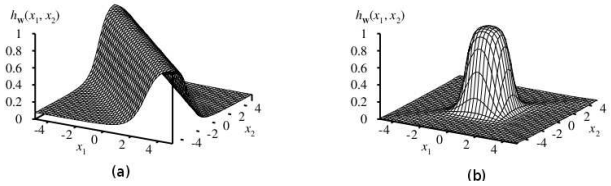

Figure 7: (a) A ridge obtained by combining two soft threshold functions. (b) The result of combining two ridges

If we were to add even more hidden units, we could produce more bumps of different sizes in different places. It is actually possible to represent any continuous function with a single, sufficiently large hidden layer, and with two layers we can even represent discontinuous functions.

Learning algorithms for multi-layer networks are fairly similar to the perceptron learning algorithm. One difference, however, is that we may have several outputs, so we have an output vector $h_w(x)$, rather than a single value, and each example has an output vector y. An even larger difference is that, while the error at the output layer is clear, the error at the hidden layers is rather confusing, as the training data does not say what value the hidden nodes should have. It turns out that we must back-propagate the error from the output layer to the hidden layers.

At the output layer, the weight-update rule is identical to Equation 7. We have multiple output units, so let Err_i be the ith component of the error vector. We will also define a modified error $\Delta_i = Err_i \times g'(in_i)$ so that the weight-update rule becomes:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \tag{8}$$

To update the connections between the inputs and the hidden units, we will need to define a quantity that is analogous to the error term for the output nodes. Here is where the error back-propagation will occur. The idea is that the

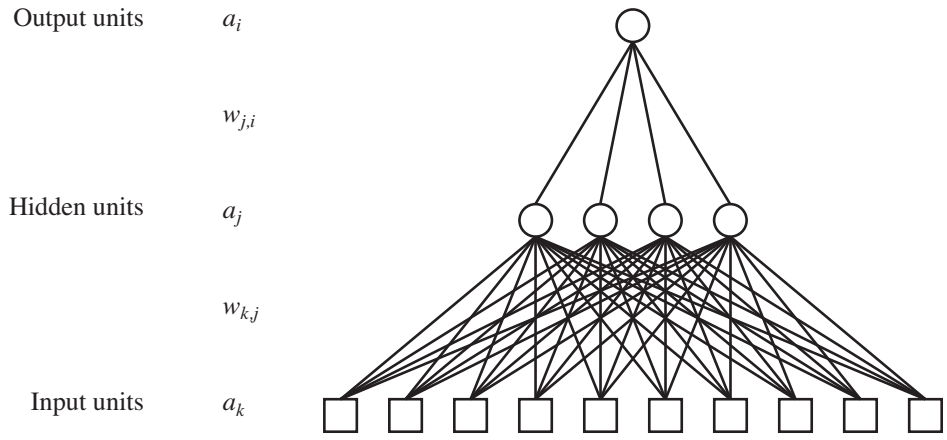

Figure 8: A multi-layer neural network with 10 inputs, 4 hidden units, and 1 output

hidden node j is responsible for some fraction of the error Δ_i in each of the output nodes that it is connected to. Thus, the Δ_i values are divided according to the strength of the connection between the hidden node and the output node, and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for these values is as follows:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \tag{9}$$

Now the weight-update rule for the weights between the inputs and the hidden layer is almost identical to the update rule for the output layer. It becomes:

$$W_{k,j} \leftarrow W_{k,j} + \alpha + a_k + \Delta_j$$

The back-propagation process can then be summarized as follows:

1. Compute the Δ values for the output units, using the observed error.
2. Starting with the output layer, repeat the following for each layer in the network, until the earliest hidden layer has been reached:
   - Propagate the Δ values back to the previous layer.
   - Update the weights between the two layers

4 Conclusions

Neural networks can be used for complex learning tasks. However, they do have the disadvantage that a certain amount of fiddling around is needed in order to get a network structure which is appropriate for the problem at hand, and which will achieve convergence to something close to the global optimum in the weight space.