Notes on Multilayer, Feedforward Neural Networks

CS425/528: Machine Learning
Fall 2012

Prepared by: Lynne E. Parker

[Material in these notes was gleaned from various sources, including E. Alpaydin's book *Introduction to Machine Learning*, MIT Press, 2004; and T. Mitchell's book *Machine Learning*, McGraw Hill, 1997.]

I. Purpose of Artificial Neural Networks

An *artificial neural network* (ANN) (or, more simply, *neural network* or *neural net*) provides a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Neural network learning is a type of *supervised* learning, meaning that we provide the network with example inputs and the correct answer for that input. Neural networks are commonly used for *classification* problems and *regression* problems. In *classification* problems, the objective is to determine which class (out of several possibilities) that an input belongs to. For example, say we want to have a network learn to distinguish pictures of cats from pictures of dogs. So, we would provide the network with a series of pictures, and for each picture, we would tell the network whether the picture is of a cat or a dog. In *regression* problems, the objective is to learn a real-valued target function. An example would be to learn the relationship between economy metrics (such as GDP, unemployment rate, inflation rate, average personal savings, etc.) and the stock market performance. The type of problem you're trying to solve (i.e., whether you're solving a regression problem or a classification problem) will determine exactly how you structure your network. More on this later.

II. Network Structure

A neural network consists of layers of interconnected "artificial neurons", as shown in Figure 1. A "neuron" in a neural network is sometimes called a "node" or "unit"; all these terms mean the same thing, and are interchangeable.

A multilayer feedforward neural network consists of a layer of input units, one or more layers of hidden units, and one output layer of units. A neural network that has no hidden units is called a *Perceptron*. However, a perceptron can only represent linear functions, so it isn't powerful enough for the kinds of applications we want to solve. On the other hand, a multilayer feedforward neural network can represent a very broad set of nonlinear functions[1]. So, it is very useful in practice.

The most common network structure we will deal with is a network with one layer of hidden units, so for the rest of these notes, we'll make the assumption that we have exactly one layer of hidden units in addition to one layer of input units and one layer of output units. This structure is called *multilayer* because it has a layer of processing units (i.e., the hidden units) in addition to the output units. These networks are called *feedforward* because the output from one layer of neurons feeds forward into the next layer of neurons. There are never any backward connections, and connections never skip a layer. Typically, the layers are *fully connected*, meaning that all units at one layer are connected with all units at the next layer. So, this means that all input units are connected to all the units in the layer of hidden units, and all the units in the hidden layer are connected to all the output units.

Usually, determining the number of input units and output units is clear from your application. However, determining the number of hidden units is a bit of an art form, and requires experimentation to determine the best number of hidden units. Too few hidden units will prevent the network from being able to learn the required function, because it will have too few degrees of freedom. Too many hidden units may cause the network to tend to overfit the training data, thus reducing generalization accuracy. In many applications, some minimum number of hidden units is needed to learn the target function accurately, but extra hidden units above this number do not significantly affect the generalization accuracy, as long as cross validation techniques are used (described later). Too many hidden units can also significantly increase the training time.

Each connection between nodes has a *weight* associated with it. In addition, there is a special weight (called w_0) that feeds into every node at the hidden layer and a special weight (called z_0) that feeds into every node at the output layer. These weights

[1] Research is ongoing to determine exactly which functions are learnable by ANNs; for our purposes, we just need to know that multilayer feedforward neural networks can express most nonlinear functions that we care about.

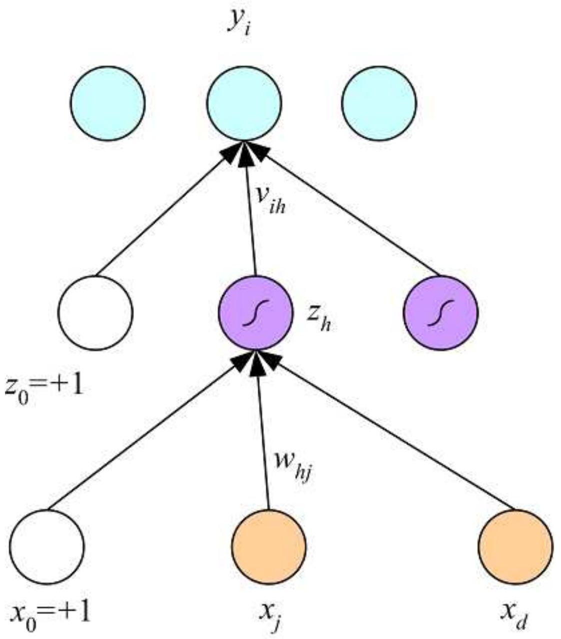

Fig. 1. Typical structure of a multilayer feedfoward artificial neural network. Here, there is one layer of input nodes (shown in the bottom row), one layer of hidden nodes (i.e., the middle row), and one layer of output nodes (at the top). The number of nodes per layer is application-dependent.

are called the *bias*, and set the thresholding values for the nodes. We'll come back to this later. Initially, all of the weights are set to some small random values near zero. The training of our network will adjust these weights (using the Backpropagation algorithm that we'll describe later) so that the output generated by the network matches the correct output.

III. Processing at a node

Every node in the hidden layer and in the output layer processes its weighted input to produce an output. This can be done slightly differently at the hidden layer, compared to the output layer. Here's how it works.

A. Input units

The input data you provide your network comes through the input units. No processing takes place in an input unit – it simply feeds data into the system. For example, if you are inputting a grayscale image (e.g., a grayscale picture of your pet Fido) to your network, your picture will be divided into pixels (say, 120 x 128 pixels), each of which is represented by a number (typically in the range from 0 to 255) that says what the grayscale value is for that piece of the image. One pixel (i.e., a number from 0 to 255) will be fed into each input unit. So, if you have an image of 120 x 128 pixels, you'll have 15360 input units[2].

The value coming out of an input unit is labeled x_j, for j going from 1 to d, representing d input units. There is also a special input unit labeled x_0, which always has the value of 1. This is used to provide the bias to the hidden nodes. (More on this soon.)

B. Hidden units

The connections coming out of an input unit have weights associated with them. A weight going to hidden unit z_h from input unit x_j would be labeled w_{hj}. The bias input node, x_0, is connected to all the hidden units, with weights w_{h0}. In the training, these bias weights, w_{h0}, are treated like all other weights, and are updated according to the backpropagation algorithm we'll discuss later. Remember, the value coming out of x_0 is always 1.

Each hidden node calculates the weighted sum of its inputs and applies a thresholding function to determine the output of the hidden node. The weighted sum of the inputs for hidden node z_h is calculated as:

$$\sum_{j=0}^{d} w_{hj} x_j \tag{1}$$

[2] Often, you'll preprocess your input to reduce the number of input units in your network. So, in this case, you might average several pixels to reduce the resolution down to, say, 30 x 32, which is just 96 input units.

The thresholding function applied at the hidden node is typically either a step function or a sigmoid function. For our purposes, we'll stick with the sigmoid function. The general form of the sigmoid function is:

$$\text{sigmoid}(a) = \frac{1}{1+e^{-a}} \tag{2}$$

The sigmoid function is sometimes called the "squashing" function, because it squashes its input (i.e., a) to a value between 0 and 1. At the hidden node, we apply the sigmoid function to the weighted sum of the inputs to the hidden node, so we get the output of hidden node z_h is:

$$z_h = \text{sigmoid}\left(\sum_{j=0}^{d} w_{hj}x_j\right) = \frac{1}{1+e^{-\sum_{j=0}^{d} w_{hj}x_j}} \tag{3}$$

for h going from 1 to H, where H is the total number of hidden nodes.

C. Output units

Now, we can do a similar computation for the output nodes. The difference is that the exact way we compute our output depends on the type of problem we're solving – either a regression problem or a classification problem. And, the calculation also depends on whether we have 1 output unit or multiple output units.

We start out the same as we did with the hidden units, calculating the weighted sum. We label the weights going into output unit i from hidden unit h as v_{ih}. Just like the input layer, we also have a bias at the hidden layer. So, each output unit has a bias input from hidden unit z_0, where the input from z_0 is always 1 and the weights associated with that input are trained just like all the other weights.

So, output unit i computes the weighted sum of its inputs as:

$$o_i = \sum_{h=0}^{H} v_{ih}z_h \tag{4}$$

If there is just one output unit, then we omit the i subscripts, and we have:

$$o = \sum_{h=0}^{H} v_h z_h \tag{5}$$

Now, we have to decide what function we're going to apply to this weighted sum to generate y_i, which is the output of unit i. We'll look at four cases:

1) **Regression with a single output.** This is the problem of learning a function, where the single output corresponds to the value of the function for the given input. Here, because we are learning a function, we do not want our output to be "squashed" to be between 0 and 1 (which is what the sigmoid function does). Instead, we just want the regular, unthresholded output. So, in this case, we calculate the output unit value of y as simply the weighted sum of its inputs:

$$y = o = \sum_{h=0}^{H} v_h z_h \tag{6}$$

Note here that we call this y instead of y_i because we only have 1 output unit.

2) **Regression with multiple (i.e., K) outputs.** This is the regression problem applied to several functions at once; that is, we learn to approximate several functions at once, and each output corresponds to the output of one of those functions. Similar to the previous case, we don't want to squash the output. However, here, we have multiple output nodes. So, we calculate the output value of unit y_i as:

$$y_i = o_i = \sum_{h=0}^{H} v_{ih}z_h \tag{7}$$

3) **Classification for 2 classes.** This is the problem of discriminating between two classes. Since one node can output either a 0 or a 1, we can have one class correspond to a 0 output, and the other class correspond to an output of 1. Here, we do want our output to be squashed between 0 and 1. So, we apply the sigmoid function at our output unit, y, to get the following output:

$$y = \text{sigmoid}(o) = \text{sigmoid}\left(\sum_{h=0}^{H} v_h z_h\right) = \frac{1}{1+e^{-\sum_{h=0}^{H} v_h z_h}} \tag{8}$$

4) **Classification for $K > 2$ classes.** Here, we typically have K output nodes, for K classes. We usually have one ouput node per class, instead of having $log_2(K)$ output nodes for a couple of reasons. First, our network will have a more expressive

space to find a function when we have more weights to learn. Second, with this structure, we can get information on the second choice of the network (e.g., if the first output node gives value of 0.9 and a second node gives a value of 0.8, we know that the network doesn't have strong "confidence" that the first class is the correct class, since 0.9 and 0.8 are both high values. We would know the network had high confidence if only one of the outputs were close to 1, and the rest were close to 0.)

In this case, we do want our output values to be between 0 and 1, so we could apply the sigmoid function at each of the output nodes. However, we still have one more step to generate the ultimate answer of the network, which is to determine which of the output nodes has the largest value. Whichever node generates the largest value tells us which class the network believes the input belongs to. We could do this by applying the max function to the outputs. However, a nicer way of doing this is to apply the "softmax" function to the weighted sum. The softmax function has the effect of making the maximum value of the outputs to be close to 1 and the rest to be close to 0. An added bonus is that it is differentiable, which is nice for some theoretical proofs (but which we'll skip!). So, we'll calculate the output of node y_i as:

$$y_i = \text{softmax}(o_i) = \frac{e^{o_i}}{\sum_{i=1}^{K} e^{o_i}} = \frac{e^{\sum_{h=0}^{H} v_{ih} z_h}}{\sum_{i=1}^{K} e^{\sum_{h=0}^{H} v_{ih} z_h}} \tag{9}$$

Note that calculating the softmax function is a 2-step process – you first have to calculate the o_i values of each output node, and then you apply the softmax function to each output node.

IV. Training the Network

Training your neural network to produce the correct outputs for the given inputs is an iterative process, in which you repeatedly present the network with an example, compare the output on this example (sometimes called the *actual output*) with the desired output (sometimes called the *target output*), and adjust the weights in the network to (hopefully) generate better output the next time (i.e., output that is closer to the correct answer). By training the network over and over with various examples, and using the Backpropagation algorithm (which we'll talk about in a minute) to adjust the weights, the network should learn to produce the correct answer. Ideally, the "correct answer" is not just the right answer for the data that you train your network on, but also for generalizations of that data. (For example, if you train your network on pictures of all the cats in your neighborhood, you still want it to recognize Morris (the 9 Lives cat, remember?) as a cat, not a dog. Hmmm ... I'm assuming Morris doesn't live in your neighborhood.)

You train your network using a data set of examples (called *training data*). For each example, you know the correct answer, and you tell the network this correct answer. We will call the process of running 1 example through your network (and training your network on that 1 example) a *weight update iteration*. Training your network once on each example of your training set is called an *epoch*. Typically, you have to train your network for many epochs before it *converges*, meaning that the network has settled in on a function that it thinks is the best predictor of your input data. More about convergence later.

A. Backpropagation Algorithm

The algorithm we'll use to train the network is the Backpropagation Algorithm. The general idea with the backpropagation algorithm is to use gradient descent to update the weights so as to minimize the squared error between the network output values and the target output values. The update rules are derived by taking the partial derivative of the error function with respect to the weights to determine each weight's contribution to the error. Then, each weight is adjusted, using gradient descent, according to its contribution to the error. We won't go into the actual derivations here – you can find that in your text and in other sources. This process occurs iteratively for each layer of the network, starting with the last set of weights, and working back towards the input layer (hence the name *backpropagation*).

1) Offline versus Online Learning: Before we get to the details of the algorithm, though, we need to make clear a distinction between "offline" and "online" learning. "Offline" learning, in the context of this discussion, occurs when you compute the weight updates after summing over *all* of the training examples. "Online" learning is when you update the weights after *each* training example. The theoretical difference between the two approaches is that offline learning implements what is called *Gradient Descent*, whereas online learning implements *Stochastic Gradient Descent* (also called *Incremental Gradient Descent*).

The general approach for the weight updates is the same, whether online or offline learning is used. The only difference is that offline learning will sum the error over all inputs, while the online learning will compute the error for each input (one at a time).

2) Online Weight Updates: As with the output calculation, the weight update calculation depends on the type of problem we're trying to solve. Remember, we're looking at 4 cases, and we're calculating the weight updates given a single instance (x^t, r^t), where x^t is the input, r^t is the target output, and y^t is the actual output of the network. Here, the t superscript just means the current example that the network is training on. In these weight updates, we also use a positive constant *learning* *rate*, η, that moderates the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1), and is sometimes made to decay as the number of weight-tuning iterations increases.

Here are the 4 situations:

1) **Regression with a single output.** The weight updates for this case are:

$$\begin{aligned} \Delta v_h &= \eta(r^t - y^t)z_h^t & (10)\\ \Delta w_{hj} &= \eta(r^t - y^t)v_h z_h^t(1 - z_h^t)x_j^t & (11)\end{aligned}$$

2) **Regression with multiple (i.e., K) outputs.** The weight updates for this case are:

$$\begin{aligned} \Delta v_{ih} &= \eta(r_i^t - y_i^t)z_h^t & (12)\\ \Delta w_{hj} &= \eta\left(\sum_{i=1}^{K}(r_i^t - y_i^t)v_{ih}\right) z_h^t(1 - z_h^t)x_j^t & (13)\end{aligned}$$

3) **Classification for 2 classes.** The weight updates for this case are:

$$\begin{aligned} \Delta v_h &= \eta(r^t - y^t)z_h^t & (14)\\ \Delta w_{hj} &= \eta(r^t - y^t)v_h z_h^t(1 - z_h^t)x_j^t & (15)\end{aligned}$$

4) **Classification for $K > 2$ classes.** The weight updates for this case are:

$$\begin{aligned} \Delta v_{ih} &= \eta(r_i^t - y_i^t)z_h^t & (16)\\ \Delta w_{hj} &= \eta\left(\sum_{i=1}^{K}(r_i^t - y_i^t)v_{ih}\right) z_h^t(1 - z_h^t)x_j^t & (17)\end{aligned}$$

3) The Algorithm: Let's pick the classification problem for $K > 2$ cases as our example; then the algorithm for online backpropagation for this problem is given as Algorithm 1. This equation implements equation 9 for the output calculation, and equations 16 and 17 for the weight updates. You can change this algorithm to handle the other types of problems (i.e., regression or classification with 2 classes) by replacing the algorithm's output calculations on lines 1.8 - 1.12 with one of the equations 6, 7, or 8, as well as the weight updates, replacing the weight updates on lines 1.15 and 1.20 with the appropriate equations from subsection IV-A2.

B. Calculating the Error

Determining how well your network is approximating the desired output requires that you measure the error of the network. Many error functions are possible, but the most common error function used is the sum of squared errors. We can measure this error for one output unit for one training example (x^t, r^t) as:

$$E(\text{W,v}|x^t, r^t) = \frac{1}{2}(r^t - y^t)^2 \tag{18}$$

This equation says, "the error for a set of weights W and v, given a particular example (x^t, r^t) is one-half the sum of the squared difference between the desired output and the actual output."

When we have multiple output units, we simply sum the error over all the output units, as follows (again, for just 1 training example):

$$E(\text{W,v}|x^t, r^t) = \frac{1}{2}\sum_{i=1}^{K}(r_i^t - y_i^t)^2 \tag{19}$$

If we want to calculate the error for one output node (say, output y_i) for 1 complete epoch (i.e., for one pass of all the training examples, we simply sum the error for that output unit over all training examples, as follows:

$$E(\text{W,v}|\mathcal{X}) = \frac{1}{2}\sum_{(x^t,r^t)\in\mathcal{X}}(r_i^t - y_i^t)^2 \tag{20}$$

If we want to calculate the error for the entire network for 1 complete epoch (i.e., for one pass of all the training examples), we simply sum the error over all output units over all training examples, as follows:

$$E(\text{W,v}|\mathcal{X}) = \frac{1}{2}\sum_{(x^t,r^t)\in\mathcal{X}}\left(\sum_{i=1}^{K}(r_i^t - y_i^t)^2\right) \tag{21}$$

When you are calculating the error during validation, you would use the above equation 21 for determining the error for the network for all of the validation set.

```
Initialize all $v_{ih}$ and $w_{hj}$ to rand(-0.01,0.01);
repeat
    for all $(x^t, r^t) \in \mathcal{X}$ in random order do
        for $h = 1$ to $H$ do
            $z_h = \text{sigmoid}\left(\sum_{j=0}^{d} w_{hj} x_j^t\right)$;
        end
        $total = 0$;
        for $i = 1$ to $K$ do
            $o_i = \sum_{h=0}^{H} v_{ih} z_h)$;
            $total = total + e^{o_i}$;
        end
        for $i = 1$ to $K$ do
            $y_i = \frac{e^{o_i}}{total}$;
        end
        for $i = 1$ to $K$ do
            for $h = 0$ to $H$ do
                $\Delta v_{ih} = \eta(r_i^t - y_i^t) z_h$;
            end
        end
        for $h = 1$ to $H$ do
            for $j = 0$ to $d$ do
                $\Delta w_{hj} = \eta \left(\sum_{i=1}^{K} (r_i^t - y_i^t) v_{ih}\right) z_h (1 - z_h) x_j^t$;
            end
        end
        for $i = 1$ to $K$ do
            for $h = 0$ to $H$ do
                $v_{ih} = v_{ih} + \Delta v_{ih}$ ;
            end
        end
        for $h = 1$ to $H$ do
            for $j = 0$ to $d$ do
                $w_{hj} = w_{hj} + \Delta w_{hj}$ ;
            end
        end
    end
until converged ;
```

Algorithm 1: Online Backpropagation Algorithm for the classification problem of $K > 2$ cases.

C. Determining Convergence

The best way to determine whether your network has reached the best set of weights for your training data is to validate the results using a validation set of data. This is a separate data set that you do not use during training. Instead, you use the validation data to detect when your network is beginning to *overfit* to the training data. Remember, you want your network to generalize its inputs, so that it can correctly answer regression or classification queries not only for your training data, but also for other examples. If you train your network too long, it will overfit to the training data, which means that it will correctly answer only examples that are in your training data set.

So, to help ensure that your network does not overfit, you can use a *cross validation* procedure. This involves training your network for a while (i.e., several epochs), and then presenting your network with the validation data. Again, the validation data is a set of data that your network has never seen before. You keep track of the error of your network as it is presented with the validation data using equation 21, but you DO NOT update the network weights during this procedure.

You repeat the process of training and validating, keeping track of the errors in both cases, and you declare convergence when your validation error is consistently growing. As you go, you need to save the weights of your network when the validation error is decreasing. This way, once the validation errors begin going up, you'll have the weights to go back to that give you the best performance. Keep in mind that it is possible for the validation error to grow for a while, but then begin decreasing again (before perhaps going up again). So, you have to analyze the trend of your validation error to convince yourself that it is consistently growing before you halt the training of your network. Once you've declared convergence, you use your saved weights (i.e., from when the validation error was the lowest) for your final network.

If you have enough data, one easy way to handle the validation is to divide your dataset into 3 parts. The first part is used for training, the second part is used for validation, and the third part is used at the very end to see how well your network has truly learned, on examples it has never seen before. However, there are other ways of validating your network, too, such as $k-$fold cross validation. This approach is especially good when you have a small data set. In this approach, you divide your data set into k pieces, and save one of these pieces for validation while you train on the other $k-1$ pieces. You keep track of how many iterations it takes to converge. Then, you repeat this process for all k pieces (selecting a different piece for validation each time), keeping track of the number of iterations to convergence each time. After all k training stages are complete, you compute the average number of iterations to converge over the k stages, which we'll call $\bar{i}$. Finally, you train one last time on all the data for $\bar{i}$ iterations, and this is your final network.

D. Momentum

Gradient descent is generally a slow process, taking a long time to converge. One easy way to speed the learning process is to use *momentum*. Momentum takes into account the previous weight update when making a current weight update. So, you must save the updates made for each weight for 1 time step. Then, on the next iteration of weight updates, you make use of this previous update information. Recall that our old weight updates were as follows:

$$\begin{aligned} v_{ih} &= v_{ih} + \Delta v_{ih} \\ w_{hj} &= w_{hj} + \Delta w_{hj} \end{aligned}$$

So, to add momentum, your new weight update equations become:

$$v_{ih}^t = v_{ih}^t + \Delta v_{ih}^t + \alpha \Delta v_{ih}^{t-1} \quad (22)$$

$$w_{hj}^t = w_{hj}^t + \Delta w_{hj}^t + \alpha \Delta w_{hj}^{t-1} \quad (23)$$

Here, the superscript t refers to the current training example and $t-1$ refers to the previous training example. So, with momentum, you just add α times the previous update when adjusting your weights. Here, α is a constant called *momentum*, with $0 \leq \alpha < 1$.