Backpropagation in Feedforward Networks

Dr. Olivia Guest

October 21, 2015

1 Overview

Last week we ran and trained a 2-layer perceptron. For reasons that became obvious (i.e., if no linear separation of the target states can be accomplished) 2-layer perceptrons are not as useful as one might initially assume. So we must turn to adding a 3rd layer of units between the input and the output units. This as you might expect is called the hidden layer because they are neither input nor output units — their activations are not directly in contact with the environment the network is in. Adding hidden units produces a network called a 3-layer perceptron. As we shall see it can now be taught classification problems like exclusive-or which are not linearly separable.

Backpropagation in a feedforward network works by running the network **forwards** (as we did with the 2-layer perceptron) and then running the network **backwards**. This second phase is required because we need to calculate targets not only for the output units but also for the hidden units. Their targets are calculated based on the output units' activations and the output targets. Using these hidden unit targets we can then calculate the error of the hidden units and subsequently update the weights from the input units to the hidden units, as well as being able to update the weights from the hidden units to the output units. In other words, running the network backwards allows us to calculate errors for every single unit (apart from those in the input layer, which have no targets). The errors for every unit are then used to calculate the connection weight updates, thus adjusting our network to conform to the required output states given an input.

2 Forwards Phase: Propagation of Activations

During the feedforward phase, each unit, apart from the input units, receives activations from units in the previous layer. Thus, the input to each unit, i, is:

$$\eta_i = \sum_j s_j w_{ji} + b_i \tag{1}$$

where b_i is the bias of unit i, and s_j is the state of a unit j that projects onto i. When unit i receives input η_i, its state changes to:

$$s_i = \frac{1}{1 + e^{-\eta_i}} \tag{2}$$

as usual by means of the **logistic function** (previously we called the activation function f, when discussing the 2-layer perceptron, and we did not use the logistic function). The logistic function is the activation function that is most commonly used with backpropagation, although $tanh(\eta_i)$ (the hyperbolic tangent function) is also common.

In Figure 1, we can see a graphical depiction of what the sigmoid activation function, in this case a logistic curve. The values on the x axis represent the values that η_i (the pre-synaptic input to unit i) takes on, while the y axis represents the output of the activation function that will be the state of the unit: s_i (the post-synaptic state).

1. What does this function do to the pre-synaptic input η_i (as calculated in Equation 1)?
2. What does a high value of b_i do to unit i's post-synaptic state? What about a low bias?
3. How is this different to how we defined our activation function f last week?
4. What is the constant e in Equation 2?
5. What are the differences and similarities between running the 3-layer and the 2-layer perceptrons?
6. What type of flow control statement is appropriate for calculating the $\sum$ in Equation 1?
7. Fill in the `Feedforward()` function in the file `network_missing.py` to reflect any changes when compared to the 2-layer network.
8. How can you check that your code is correct? Is it? Hint: Use a very simple input (target is irrelevant because you are not training yet) and calculate yourself what a few of the states of the network should be using the above equations. Do they match up with the code you have written when you propagate the pattern?

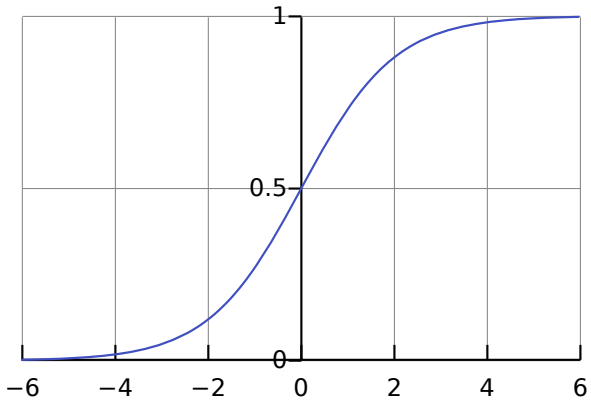

Figure 1: The shape of the logistic curve.

3 Backwards Phase: Propagation of Error Signal

After the feedforward phase, the network is run in reserve in order to gather the weight adjustments. Unit i sends the error signal:

$$\delta_i = s_i - t_i \tag{3}$$

where t_i is the target state. If i is an output unit t_i is the pre-set target pattern. This metric for calculating error is called cross-entropy error, other types (e.g., mean squared error) can also be used.

However, if i is a hidden unit, its target is:

$$t_i = \sum_j \delta_j w_{ij} \tag{4}$$

meaning that the target for hidden unit i is a function of the error signal emitted by the output units and the weights that connect them.

Now that we know how to calculate the targets for both output and hidden units, we need to calculate their respective deltas. For the output unit the error is defined in Equation 3. For hidden units we use a slightly different way of calculating the error:

$$\delta_i = s_i(1 - s_i)t_i \tag{5}$$

Now we have both types of δ_is (the errors for the output and hidden units), we may calculate the *proposed* changes to the weights:

$$\Delta w_{ij} = \delta_j s_i \tag{6}$$

These are merely suggestions to update the connection weights and do not represent the actual changes we will make, as we may decide to scale them by the learning rate, apply momentum, etc.

1. If the pre-synaptic activation of an output unit i is -6 and its target t_i is 1.0, what error signal δ_i will it emit? (Use equations 1, 2, and 3.)
2. What type of flow control statement is appropriate for calculating the $\sum$ in Equation 4?
3. What type of data is i and what type of data is s_i?
4. What are the main differences and similarities between the backwards and the feedforward phases of running a network?
5. Why is the backwards phase required?
6. How many types of target t_i are there and how do we calculate them?
7. Fill in the `Backprop()` function to run the network backwards. Remeber to run it forwards first! However, do not change the weights just yet. Merely accumulate the proposed changes to the weights by calculating Equation 6.
8. Check if your code is working correctly. Are the output errors correct, are the hidden targets correct, ...?

4 Weight Adjustments

Once the network has been run forwards and backwards for an epoch, ϵ, weight updates are calculated using:

$$w_{ij}^{\epsilon+1} = w_{ij}^{\epsilon} + \nu \Delta w_{ij}^{\epsilon-1} - \mu \Delta w_{ij}^{\epsilon} \tag{7}$$

taking into account the momentum, ν, and the learning rate, μ. Δw_{ij}^{ϵ} is the proposed weight update, from Equation 6, while $\Delta w_{ij}^{\epsilon-1}$ is the actual weight update applied last time! $\Delta w_{ij}^{\epsilon-1}$ is the change to the weights including learning rate and momentum from the previous epoch.

1. Apply the weight updates as in Equation 7. Fill in `Apply_Deltas()`. Think carefully about how you will keep track of the previous actual (not proposed) weight change $\Delta w_{ij}^{\epsilon-1}$. Remember that δ_is have been being accumulated, so they will need to be set to zero when you have applied them, in order for them to be ready to receive the new epoch's.

2. Now that your network learns, compare its performance on not with a 2-layer perceptron. Is ti faster or slower? What parameters play a role (learning rate, momentum, initial values of weights, etc.)?