Overnight:
Neural Networks and Backpropagation

Adam Selker

December 2017

1 Introduction and Learning Goals

An artificial neural network (ANN) is a computing tool made of a set of small, simple computing elements, connected together into a network. ANNs are useful in situations when it's easy to find or create training data, but it is difficult to formalize an alorithm to do what you want. An example of this is computer vision. Identifying which pictures contain cats via a normal, hand-coded algorithm would be all but impossible. On the other hand, Googling "pictures of cats" will net you as much training data as you'd like. In this overnight, we will use the example of predicting English text: given a short string, predict the next letter.

The power of a neural net can be seen both in articles like this one, and by examining living systems, i.e. your own brain. The human visual cortex is an excellent example of a neural network, and strongly resembles the multi-layered, feedforward networks we will work with here. The fact that you can read this text without thinking illustrates how effective the architecture can be.

Throughout this overnight, you will build a neural network in Python, using a few scripts that contain helper functions. You can find these at https://github.com/aselker/neural-networks-overnight. Make sure you have Python 3 installed.

By the end of this overnight, you should:

- Understand when and how neural networks are useful
- Understand how to evaluate and train a simple neural network
- Be able to implement a simple neural network from scratch in Python

Now, you've probably heard of neural networks in the context of linear algebra. While linear algebra can make things a bit easier once you know what you're doing, we won't be using it here, just to keep things simple. Everything works just as well considering individual neurons as it does vectors of them.

2 Feedforward Networks and Evaluation

Each element in a neural network is commonly called a "neuron," by analogy to a living brain. Each neuron has a number of input and output connections. At each time step, each neuron computes a single value called its "activation," and then passes that value on to all of its outputs. Each connection also has a "weight," which is a value by which it scales the signal moving through it.

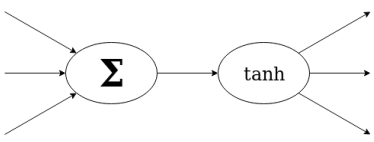


Figure 1: The inputs are summed, run through a tanh function, and then passed on to all outputs.

A neuron's activation is computed by adding together all of the neuron's inputs, and evaluating a function – the activation function – on the result. The activation functions can vary, but the one we'll be using here is the hyperbolic tangent function, tanh, which is defined as

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1}$$

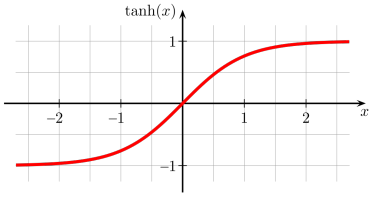


Figure 2: The hyperbolic tangent function.
Image credit: Wikimedia

This function is nice as an activation function for a few reasons. First, it is nonlinear. If the activation functions for the neurons are linear, then the whole neural network must be linear, too; this is not very useful. Another useful property of tanh is that no matter what its input, its output is always between -1 and 1. This means that the magnitudes of the signals travelling through our network should not grow too high.

Problem 1 In nn.py, implement the function 'tanh'. Use tests.py to check your answer.

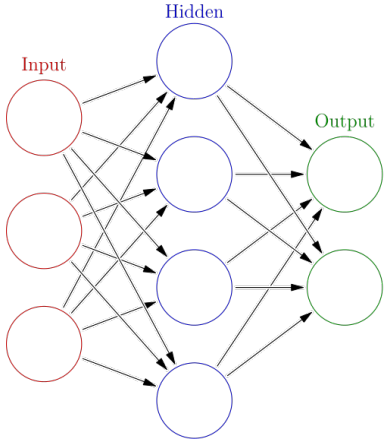


Figure 3: An ANN with three layers of neurons.
Image credit: Wikimedia

Neural networks are usually organized into layers. For clarity, we will imagine our networks as flowing from left to right, such as in Figure 3. A network with no loops in it, such as that one, is called a "feedforward network." Evaluation of such a network is performed in a few steps. First, we set all of our input neurons' activations to the values that we want as inputs – in this case, some representation of a short string of English text. Next, we move from left to right, evaluating everything in order. A connection's value is the value of its input neuron, multiplied by the connection's weight. Each neuron sums the values of its input connections, and finds the hyperbolic tangent of the resulting sum. At the end of the process, the rightmost neurons hold the output values we care about. In this case, they should hold some representation of a letter, which is the network's prediction of the letter that comes after the input string.

Problem 2 Evaluate the neural network shown in Figure 4, with the given input value and weights. What is the activation of the output neuron?

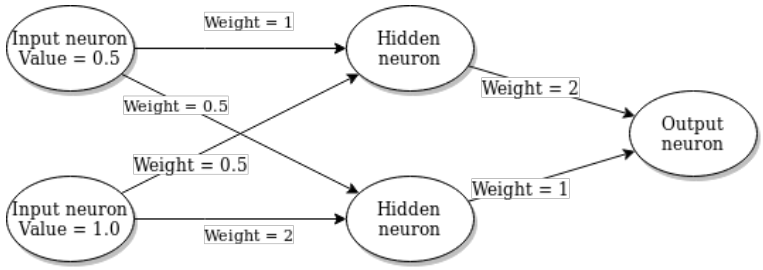


Figure 4

Problem 3 Finish the 'Neuron' class in nn.py. Everything should already be implemented, except for the 'evaluate' function. Feel free to use the tanh function you implemented earlier.

Problem 4 Using the 'Neuron' class you just made, fill in 'simpleNetwork' with an implementation of the neural network from Figure 4.

Our next step towards an English-predicting network will be to build a network which takes a single character of input, and gives a single character of output. But first, we need a way to represent letters, numbers, and punctuation as sets of input values. We will limit ourselves to 28 characters: the lowercase alphabet, the space, and the underscore. The underscore will represent any character not in our set.

One way to do this would be to assign every character a numeric value, from (say) 0 to 1, and to have a network with just one input neuron. That neuron would be set to the value corresponding to the input character. Unfortunately, this doesn't work very well. Small changes in a single neuron are hard to amplify. It is more practical to use several input neurons, each of which is set to one of just a few values.

Given that, a binary input might seem natural: convert each character to its representation in ASCII, and feed that pattern into the network. This, too, has its problems. When the binary representations of two characters are similar, the network will be biased toward considering the two inputs similar. More importantly, the network will have a harder time learning; we will explore how the learning works in the next session.

For now, we will represent every character as a series of 28 values. One, corresponding to the index of the character, will be set to 0.5; the rest will be set to 0. For instance, the letter 'c' is represented as the list

$$\begin{bmatrix} 0 & 0 & 0.5 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{2}$$

This may seem cumbersome. Fortunately, we will almost never have to deal with these lists manually; our code will do it for us. Included in nn.py are a few helper functions to make things easier.

(Why, you might ask, is the one letter set to 0.5 instead of 1? Well, these will be used for outputs as well as inputs, and the tanh function can never quite output 1 – it can get close, but it can never get there. It can, however, output 0.5.)

The next step will be to create a one-layer neural network which takes a single character, and produces another. To simplify things, there will not be a hidden layer. The connection weights will be given, and should predict the sequence "abcdabcd...".

Problem 5 Implement this neural network. Note that this should not require writing anything 28 times; we'll be getting to even bigger networks soon, so everything should be done as programmatically as possible.

3 Training and Backpropagation

So far, we have explored how to evaluate a neural network after choosing its inputs and weights. In practice, however, hand-picking weights is almost never a good idea: if that's enough to solve your problem, there's

no reason to use a neural network in the first place. The strength of neural networks is in their ability to learn.

This next part is also where things start to get complicated. If you need help, check out this video by 3Blue1Brown, or this example which is worked out fully with real numbers.

Neural networks are particularly well-suited to machine learning because they are *differentiable*. That is to say, modifying the algorithm, by adjusting the connection weights, results in a smooth change in the output. This is not the case with, for instance, a Turing machine, where changing a single instruction often causes huge changes in behavior. We can take advantage of this differentiability to slowly, incrementally modify a network to make it better.

One way to do this is through trial and error. Take a network, give it a small random change, and see if it works better. If so, keep the change. Otherwise, discard it. This can work well for small networks, but for larger networks with more complicated requirements, it is prohibitively slow. Each trial can only adjust one weight, which means that as a network grows, the time it takes to train the network will grow even faster.

Fortunately, there is a better way. Consider a neural net as a mathematical function of many variables (every input neuron and every weight) which produces several outputs (the output neurons). We can imagine defining and *error function* which combines all of the outputs into a single number representing how desirable the output is. An error function of zero generally represents the best possible output. Combining this error function with the network itself, we have a function which takes many inputs but produces only one output.

We could then take the derivative of the output of the error function with regard to any input. This would tell us in which direction, and how far, we should move that input, to make the error smaller. This way, we can change the network into a better one – train it – as directly as possible. All we have to assume is that the error function is correct, and we can make it smaller. This is the same gradient descent you used in the Neato module; it's just a bit more abstract.

(A few caveats: first, there are often *local minima*, places where the network "falls into a hole" and will not optimize its way out. There are ways of addressing this, such as adding a sort of momentum to the optimization process. Fortunately, we won't have to deal with this very much here, as our optimization criteria are fairly simple. Second, every step of the network, including the error function, must be differentiable. We'll see in a moment that this is almost always true.)

Now, it might seem complicated and slow to be taking this many derivatives. After all, every layer depends on all of the layers before it, and affects all those after it. Through clever use of the chain rule, we can evaluate the derivative with regard to every weight in about as much time as it takes to evaluate the network the normal way. The algorithm we'll use is called backpropagation, and it is one of the most important concepts in neural network research.

First, let's take a look at each component of the neural network.

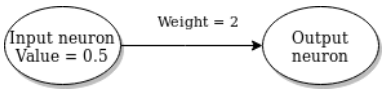
Problem 6 What is the derivative of the tanh function? Implement this as 'tanh_deriv' in nn.py.

Problem 7 The output of a connection between neurons can be modelled as the function

$$f(i, w) = i * w \tag{3}$$

where i is the activation of its input neuron, and w is the connection's weight. What is the derivative of the connection's output with regard to its weight? With regard to its input?

Problem 8 Consider this network with two neurons:



What is the derivative of the output neuron's activation, with regard to the connection's weight? With regard to the input neuron's activation?

Recall the chain rule of derivatives, which states that

$$\frac{\partial a}{\partial z} = \frac{\partial a}{\partial b} \times \frac{\partial b}{\partial c} \times \ldots \times \frac{\partial y}{\partial z} \tag{4}$$

If we have a series of functions chained upon each other, e.g.

$$h(g(f(x))) \tag{5}$$

and we know the derivative of each function, we can find the derivative of the whole system:

$$\frac{\partial h(g(f(x)))}{\partial x} = \frac{\partial h(z)}{\partial z} * \frac{\partial g(y)}{\partial y} * \frac{\partial f(x)}{\partial x} \tag{6}$$

Using this, we can find the derivatives of any part of an arbitrarily-long neural network.

Problem 9 Consider this network with three neurons:

What is the derivative of the output neuron's activation, with regard to the first connection's weight? With regard to the input neuron's activation?

This should make clear why the process is called *backpropagation*. The derivative is propagated backward, with two steps for each iteration: the derivative of the tanh function in the neuron, and the derivative of the product with the connection's weight. These mirror the way a signal is multiplied by the connections' weights, and put through the forward tanh function. The next steps are to generalize this algorithm to networks with multiple neurons in each layer, and to differentiate with respect to a reasonable error function.

Problem 10 Consider a function which sums its arguments, i.e.

$$f(a, b, c, d...) = a + b + c + d... \tag{7}$$

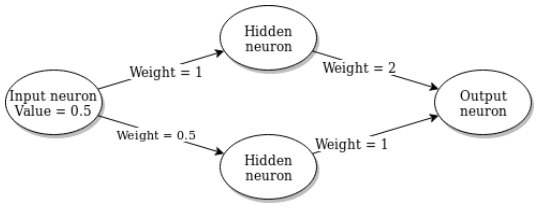
What is the derivative of this function with regard to each of its arguments?

Problem 11 One common error function, which we'll use here, is as follows:

$$error(a_o, b_o, c_o...) = \frac{1}{2} \times ((a_o - a_t)^2 + (b_o - b_t)^2 + (c_o - c_t)^2 + ...) \tag{8}$$

where a_o, b_o and so on are the outputs of the network, and a_t, b_t and so on are the respective target values. (The $\frac{1}{2}$ is to make the derivative simpler.) What is the derivative of the total error with regard to each of the network's outputs?

Problem 12 Consider this network with four neurons:



The network's target value is zero. What is the derivative of error, with regard to the input neuron's activation?

(This is the last one you'll have to evaluate by hand, we swear.)

When using backpropagation to train a network, the program usually runs the network in a loop, like this:

- Set the input neurons to whatever you'd like to feed into the network.
- Evaluate the network forwards, saving each neuron's activation.
- Calculate the network's error, and the derivative of the error with regard to every output neuron.
- Backpropagate the derivatives through the network, multiplying each by the learning rate (a number of your choosing) and subtracting them from the weights.

We're now ready to implement backpropagation in Python! We recommend adding it onto the Neuron class from before, but if you'd like to restructure, it's up to you. Note that there is no test for the backprop() function, since there are design choices to be made. There is, however, code to train and evaluate your network, which you can use to see if it's working.