NEURAL NETWORK PACKAGE MODULE 3

BACK PROPAGATION AND GRADIENT DESCENT

QMIND EDUCATE

ANDREW FARLEY

TABLE OF CONTENTS

BACK PROPAGATION

What is Propagation?

Throughout the last two modules, we have been throwing the term 'propagation' around quite a bit. What does this word actually mean and how does it apply to neural networks? The first Oxford Dictionary definition of propagation is "the act of spreading ideas, beliefs or information among many people". This doesn't directly relate to our neural network; however, there are many parallels we can draw.

When we propagate an input with *forward propagation*, we are **spreading** each input amongst **many neurons**, like how information can spread among people. The difference here is that we often won't have just one input, but many inputs. Each input will propagate itself throughout the network by using its weighted connections to other neurons. This is analogous to multiple people in a group each spreading slightly different rumors to people in another group. Every person in the latter group will draw a conclusion as to what rumor they actually believe based on their trust of the person who tells them. The trust is like the weighted connections and the rumor is the input being spread!

With forward propagation we know we propagate a given input through the network, but what is **back propagation** and what data will it propagate?

What Data Do We Feed Back Propagation?

Back propagation can be simply defined as propagating **error,** *backwards* through the network. When we train a neural network, we must first start with some labelled inputs. This means we have a list of inputs that have a **known** output value that will allow us to find the error from what the network is currently outputting and **propagate** that error back through our network. The difference between forward and back propagation is that we don't care what the 'final output' value of propagating the error to the front of the network will be in back propagation. We want to use this error at every layer to **strengthen or weaken** connections between neurons based on how much they contribute to the error. If one connection has a massive associated error, its connection will be weakened more than others to improve the overall accuracy of the system.

How do we know how much to weaken or strengthen connections? We can get the exact value to adjust connections through a process known as **gradient descent**.

GRADIENT DESCENT

Gradients

A **gradient** is a mathematical term used to describe the *rate of change* of *every variable* in a function. As you recall, we can think about our neural network as one massive function with millions, if not billions of inputs. We want to minimize the difference (or associated error) between this function and what we expect the output to be. We will use the *gradient* to constantly 'step' in the correct direction (by changing the variables) to achieve this minimum. Rather than taking the gradient of the entire function (which would take allot of computational power) we can just find the gradient at each layer and apply weight changes as necessary. The mathematics behind gradients will be discussed more in-depth next module where we talk about the mathematical theory behind Neural Networks.

Gradients and Neural Networks

So, what will the gradient look like on each layer of a neural network? We know that the gradient represents the rate of change of each variable at a certain point (or input).. This can be thought of as how much the variable needs to change to produce a more accurate output. Therefore, we must ensure the gradient encompasses every possible error in the neural network system. In our case, there are only two error sources: the error between the expected output versus the actual output (this is often calculated with a 'cost' function) and the error introduced by the activation function. There are many different cost functions which each serve their own purpose, but the easiest one (which we will use) is simply subtracting the actual output from the expected.

$$Cost = Expected - Actual$$

An Aside:

A cost function is what we use to get the initial error in our network. Remember that our entire network can be thought of a function, let's call it N(x) where x is the input matrix and N is the function that will produce our output matrix. The cost function above now translates to:

$$Cost = Expected - N(x)$$

By using calculus to optimize this function, we will reduce the error to be next to nothing making our network accurate. Cost functions can be more than just subtraction, however they will always involve some expected value and what the network is currently producing.

The way we find the error introduced by the activation function is through its derivative. The derivative is the rate of change of a function at any point which, in this case, will also be proportional to the error it produces. For the sigmoid function:

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

The derivative is:

$$f'(x) = f(x) * (1 - f(x))$$

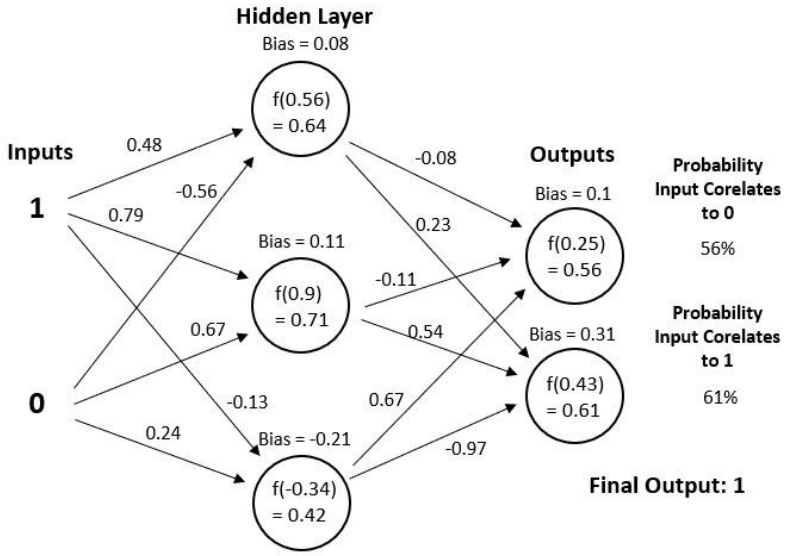
The sigmoid function is interesting as the derivative directly depends on itself! This is helpful for us because we only need to store the values each layer produces and insert them into the given formula to produce the error that sigmoid introduces into our network. If this were not the case, we would have to re-compute values from the previous layer (and possibly the entire network) which would lead to more computation time. We can then calculate the gradient at that layer with the given formula.

$$Gradient = Cost * f'$$

We must ensure that we calculate this for every neuron in the current layer by utilizing element-wise multiplication to produce a vector in the same orientation as our outputs.

Example Calculation

Let's take another look at our old friend, the 'AND' network from module one and two. We will walk through, step by step, how to compute the gradient of a given layer using real numbers. Take the network as seen in the state below:



Because our inputs are 1 and 0 our output should be 0, which means our output vector should be [1, 0]. The first thing we always need with our gradient is the result of the cost function as this is the function we need to optimize and is the starting 'input' to the backside of our network. We will use the same cost function described above.

$$Cost = Expected - Actual$$

$$Cost_{Output} = [1, 0] - [0.56, 0.61] = [0.44, -0.61]$$

Next, we need to calculate the value of the derivative of the activation function:

$$f'(x) = f(x) * \big(1 - f(x)\big)$$

$$f'_{Output} = [0.56, 0.61] *_{element} (1 - [0.56, 0.61]) = [0.56, 0.61] *_{element} [0.44, 0.39]$$

$$f'_{Output} = [0.56 * 0.44, 0.61 * 0.39] = [0.25, 0.24]$$

Notice that when we do multiplication in this, we are doing **element-wise** multiplication (denoted by $*_{element}$). This means that we do multiplication similar to how we do addition with matrices. We look at elements that are in the same place, multiply their values, and put the result in the same location. I want to be clear that this is very different from the dot product, no summation of products is happening. We do not compute the dot product because the gradient's components are specific to each neuron. We only care about the element of the gradient that is associated with its associated neuron. Great, we are all set! Let's compute the gradient for the output layer:

$$Gradient_{Output} = Cost_{Output} *_{element} f'_{Output}$$

$$Gradient_{Output} = [0.44, -0.61] *_{element} [0.25, 0.24]$$

$$Gradient_{Output} = [0.11, -0.15]$$

Notice again that we are still doing *element-wise multiplication* because each element of the gradient is associated with how much the values of its corresponding neuron are changing.

Strengthening and Weakening Connections

Great, we have calculated our gradient! But what do we do with it? Remember that the gradient will be used to *adjust the weights from the previous layer to the current layer*. At every layer during forward propagation, error is introduced by the outputs of the last layer and the weights between the layers. We are trying to calculate the error the weights brought about so we can step them in the other direction. We will do this by using the outputs from the previous layer.

The weight change for every connection should just be the last neuron's output multiplied by the gradient at the next neuron.

$$\Delta W_{between\ 2\ neurons} = f(x)_{last\ layer\ neuron} * gradient_{current\ neuron}$$

For example, if we wanted to find the weight change needed between the top two neurons (in position [0,0] from the network above, we would use the following equation:

$$\Delta W_{0,0} = f(x)_0 * gradient[0]$$

$$\Delta W_{0,0} = 0.64 * 0.11 = 0.07$$

This means we should add 0.07 to the weighting between these two neurons. If done this way the new weight would be -0.01 instead of -0.08.

$$New\ W_{0,0} = \ Old\ W_{0,0} + \Delta W_{0,0}$$

$$New\ W_{0,0} = \ -0.08 + 0.07$$

To do this for all weights in the layer we can just multiply the gradient matrix with the last layers output matrix but in a certain way. Currently, they both will be in the same orientation; however, there is a matrix operation called **transpose** which will flip a matrix along its diagonal. Having our last layer output in the shape *nx1* and our gradient in the shape *1xn* will produce a matrix the exact dimensions of the weight matrix when the dot product is performed. Having this dimension match the weight matrix is important as we can then simply add these matrices together to

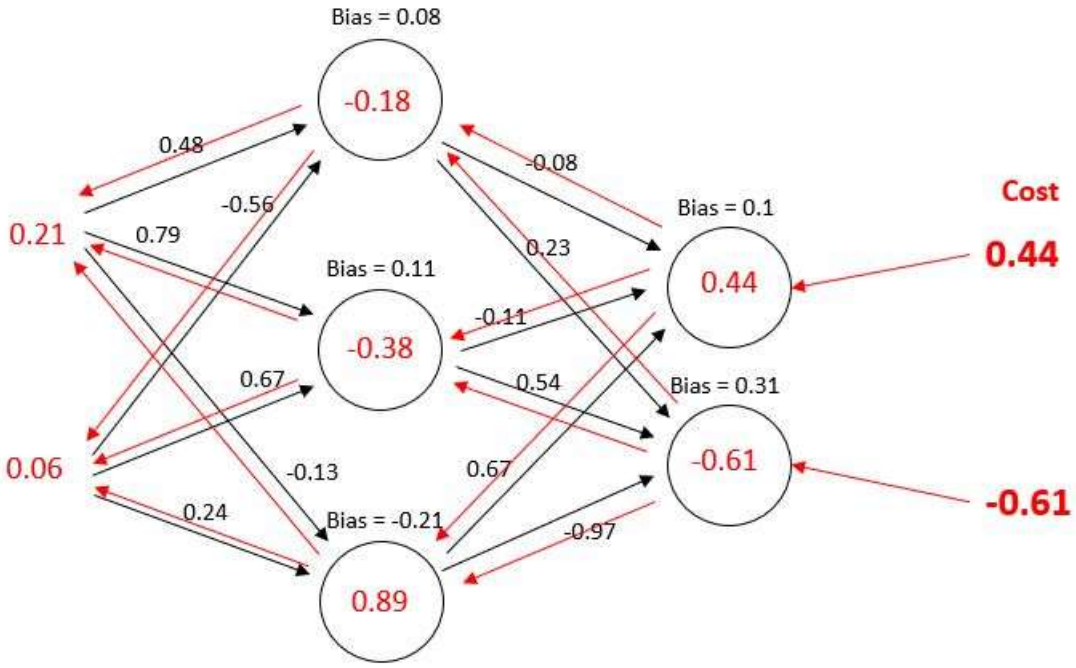
produce our new weight matrix! This is how we change the weighted connections to improve our networks performance. But what happens if the magnitude of the change is too big?

Learning Rate

Looking at the example above where we calculated the amount a weighting should change, it might have shocked you to see how big the magnitude the weight changes were. This can lead to problems as we can overstep the optimal values for our weightings and keep oscillating back and forth, never reaching a precise minimum. To account for this, we multiply every value in the gradient by something called a **learning rate**. This is a number between 0 and 1 which will reduce the overall impact that the gradient has on changing the weightings. This is good because it will allow our network to be much more accurate despite it taking longer to train due to the smaller 'steps'. A good default value for the learning rate is 0.05 which will give the network a good accuracy while not gaining too much training time. It is up to the network creator to decide on the learning rate. Different learning rates will work better for different datasets.

Error Propagation

We are still missing something key, what about the next layer? How will we change its weights? How will we get the error matrix for that layer? The answer has been previously mentioned: we will propagate the cost matrix backwards through our network. At every layer we will compute the gradient, find the weight changes, and adjust the weights. We will repeat this process until there are no more layers, meaning our network has taken a step towards better understanding our input. The process of error propagation is similar to forward propagation as we are propagating our error through every layer. You can imagine the network was flipped around and the cost matrix is now an input to this flipped network. Just perform the dot product with the transposed weight matrix (ignoring the biases) to get the cost matrix for the next layer.



The network above is an example of the cost matrix being back propagated through a Neural Net. Everything in red is how the backpropagation would be performed. We use the same weight values as we would going forward however just in reverse. Transposing the weight matrix has the effect of reversing the weights.

CLOSING REMARKS

This process of backpropagation is how most neural networks learn. There is so much more theory behind setting up training data, ensuring randomness when training, training the network in different epochs with batches of data, choosing a good cost and activation function. The list goes on and on, far outside the scope of this package. All these network properties (and more) will lead to a more accurate and effective Neural Network. This module is the last one of the theories behind how Neural Networks operate. The next module will focus on the more complex mathematics concepts covered in the last three modules and the last two modules of this package discuss implementing the theory covered in the past 3 modules in code. It is one thing to understand how a neural network operates but creating one yourself produces a host of problems and brings up many more questions. We will walk through how to program a neural network completely from scratch and then move on to using libraries such as TensorFlow and Keras. Get excited!

REFERENCES

[1] G. Sanderson, "YouTube," 3blue1brown, 5 October 2017. [Online]. Available: https://www.youtube.com/watch?v=aircAruvnKk&index=1&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. [Accessed 2018].

[2] G. Sanderson, "YouTube," 3blue1brown, 3 November 2017. [Online]. Available: https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3. [Accessed 2018].

[3] G. Sanderson, "YouTube," 3blue1brown, 16 October 2017. [Online]. Available: https://www.youtube.com/watch?v=IHZwWFHWa-w&index=2&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. [Accessed 2018].

[4] G. Sanderson, "YouTube," 3blue1brown, 3 November 2017. [Online]. Available: https://www.youtube.com/watch?v=tIeHLnjs5U8&index=4&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. [Accessed 2018].

[5] D. Shiffman, "YouTube," The Coding Train, 7 February 2018. [Online]. Available: https://www.youtube.com/watch?v=tlqinMNM4xs&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh&index=18. [Accessed 2018].

[6] D. Shiffman, "YouTube," The Coding Train, 6 February 2018. [Online]. Available: https://www.youtube.com/watch?v=qB2nwJxNVxM&index=17&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[7] D. Shiffman, "YouTube," The Coding Train, 5 February 2018. [Online]. Available: https://www.youtube.com/watch?v=8H2ODPNxEgA&index=16&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[8] D. Shiffman, "YouTube," The Coding Train, 24 January 2018. [Online]. Available: https://www.youtube.com/watch?v=r2-P1Fi1g60&index=15&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[9] D. Shiffman, "YouTube," The Coding Train, 23 January 2018. [Online]. Available: https://www.youtube.com/watch?v=QJoa0JYaX1I&index=14&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].