The Backpropagation Algorithm

Marcus Quirk

30 January 2020

1 Geron's Summary

Initialize all the hidden layers connection weights randomly.

Randomly initializing the weights breaks the symmetry and allows backpropagation to train a diverse team of neurons.

The backpropagation algorithm works well with many activation functions:

- the logistic function: $\sigma z = \frac{1}{1+e^{-z}}$
- the hyperbolic tangent function $\tanh(z) = 2\sigma(2z) - 1$
- the rectified linear unit function: $ReLu(z) = \max(0, z)$

Learn more about the hyperbolic tangent here.

The derivative of the hyperbolic tangent is the square of the hyperbolic secant. Learn more about that function here.

1.1 Short Version of Backpropagation Algorithm

For each training instance...

- the backpropagation algorithm first makes a prediction (forward pass) and measures the error
- then goes through each layer in reverse to measure the error contribution from each connection (reverse pass),
- and finally tweaks the connection weights to reduce the error (Gradient Descent step).

1.2 Long Version of Backpropagation Algorithm

- It handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an epoch.
- Each mini-batch is passed to the networks input layer, which sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the forward pass: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
- Next, the algorithm measures the networks output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output connection contributed to the error. This is done analytically by applying the chain rule (perhaps the most fundamental rule in calculus), which makes this step fast and precise.
- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

1.3 Code

```
import numpy as np

v = np.array( [3, 4] )

print( "v = ", v )

dot_product = v.dot( v )

print( "dot product = ", dot_product )

magnitude = np.sqrt( v.dot( v ) )
```

```
print( "magnitude = ", magnitude )

base_natural_log = np.e

print( "base of natural logarithms = ", base_natural_log )

print( "base of natural logarithms = ", np.exp( 1 ) )

print( "tanh( 0 ) = ", np.tanh( 0 ) )

print( "tanh( 1 ) = ", np.tanh( 1 ) )

print( "tanh( -1 ) = ", np.tanh( -1 ) )

print( "tanh( 5 ) = ", np.tanh( 5 ) )

print( "tanh( -5 ) = ", np.tanh( -5 ) )

sigmoid = lambda x: 1 / (1 + np.exp( - x ))

print( "sigmoid(0) = ", sigmoid( 0 ) )

print( "sigmoid(5) = ", sigmoid( 5 ) )

print( "sigmoid(-5) = ", sigmoid( -5 ) )

w = np.array( [np.random.random() for i in range(10)] )

print( "array of 10 random numbers = ", w )
```

2 Introduction

Multilayer perceptrons (MLPs) is a form of artificial neural network used in modern-day Machine Learning. Their design is inspired by the neural networks of the human brain. MLPs consist of layers of threshold logic units (TLUs). In the MLP's neural network, the TLU is the equivalent of a single neuron.

Each TLU in a MLP is, at its most elementary level, a computer of a weighted sum of its inputs. It accepts a number of inputs and multiplies each input value by a certain weight.

In order for an MLP to be used in a machine learning model, it needs to be set up with the correct weights at each TLU in order to make acceptable predictions. In order to determine the weights to use, an algorithm called the backpropagation algorithm is used.

The algorithm involves making a prediction (forward pass) and calculates the error with a loss function, then goes backwards through the layers to find how much each weight contributed to the overall error. From this insight, the algorithm then tweaks the connection weights to reduce the error. This process is repeated many times over, with each pass being known as an 'epoch'.

In this project, I would like to attempt to apply the backpropagation algorithm by hand for myself. My project from last week (attempting a machine learning project from scratch) definitely contributed to my increased and consolidated understanding of the material in the textbook. My hope is that doing similarly for this algorithm not only helps me understand backpropagation, but also give me some insight to a more natural understanding of basic neural networks.

Currently, I understand how MLPs and neural networks work on a theoretical level, but my intuitive understanding is lacking. By exploring this network in-depth, I think that I will better intrinsically understand how neural networks work. I believe that this could also help me in the future to understand whether or not a neural network would be an appropriate choice for whatever machine learning project I take on.

I will be using a simple MLP with two inputs and one bias. The middle (or "hidden") layer consists of two TLUs and the output layer has two TLUs; there are two inputs and two outputs.

Input	**Desired Output**
0.56	0.08
0.24	0.83

Let **IN** be a vector that contains the input values plus a bias. The bias is equal to one.

$$\mathbf{IN} = \begin{bmatrix} 0.56 \\ 0.24 \\ 1.00 \end{bmatrix}$$

Let **OUT** be a vector that contains the output values.

$$\mathbf{OUT} = \begin{bmatrix} 0.08 \\ 0.83 \end{bmatrix}$$

It is important to randomise the initial connection weights, otherwise training will not work. For example, if all the weights are equal to begin with, every neuron in every layer would be identical, and backpropagation apply the same change to every neuron. The model would not work. I used random.org to get

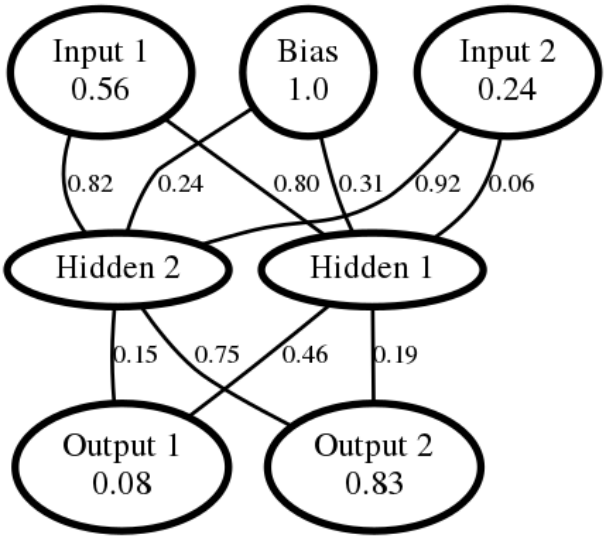

Figure 1: Multi-Layer Perceptron

my weights (use the Decimal Fraction Generator under FREE Services/Numbers):

Here are your random numbers:

$$\mathbf{W} = \begin{bmatrix} 0.80 \\ 0.82 \\ 0.06 \\ 0.92 \\ 0.31 \\ 0.24 \\ 0.46 \\ 0.19 \\ 0.15 \\ 0.75 \end{bmatrix}$$

3 Forward Pass

The basic equation for the weighted sum of the inputs can be expressed as the following, for the first layer of neurons in the MLP:

$$h_1 = \mathbf{W}^T \cdot \mathbf{IN}$$
$$h_1 = w_1 \cdot in_1 + w_2 \cdot in_2 + w_3 \cdot b$$

Where in_1 and in_2 are the two inputs, b is the bias, and w_1, w_2, and w_3 are the corresponding weights. In this example:

$$\begin{aligned} h_1 &= (0.80 \cdot 0.56) + (0.06 \cdot 0.24) + (0.31 \cdot 1) \\ &= 0.448 + 0.0144 + 0.31 \\ &= 0.7724 \end{aligned}$$

Every MLP uses an activation function at each neuron. The textbook by 'default' uses the logistic function because it has a nonzero derivative for every value, as opposed to an activation function such as the step function which evaluates to -1 for any negative value and $+1$ for any positive value.

It is important for an activation function to be differentiable at every value because we will be finding the derivative of the function during the backwards pass of the backpropagation algorithm later. Therefore, I will select the hyperbolic tangent function (tanh) as my activation function. Any input value to tanh evaluates to an output between -1 and $+1$ (instead of 0 and 1 as with the default logistic function), and it is a continuous function (therefore differentiable for all values unlike the step function).

We apply our activation function:

$$\tanh(0.7724) = 0.6483$$

The final output of the hidden neuron 1 is, therefore, 0.6483.

We can do the same for hidden neuron 2:

$$\begin{aligned} h_2 &= (0.82 \cdot 0.56) + (0.92 \cdot 0.24) + (0.24 \cdot 1) \\ &= 0.92 \end{aligned}$$

$$\tanh(0.92) = 0.7259$$

We repeat the process for the output neurons. The inputs for this layer are the outputs of the hidden neurons in the previous layer; recall that the output for h_1 is 0.6483 and the output for h_2 is 0.7259. There is also no bias at this layer.

$$
\begin{aligned}
o_1 &= (0.46 \cdot 0.6482) + (0.15 \cdot 0.7259) \\
&= 0.407057
\end{aligned}
$$

$$\tanh(0.407057) = 0.3860$$

$$
\begin{aligned}
o_2 &= (0.19 \cdot 0.6482) + (0.75 \cdot 0.7259) \\
&= 0.667583
\end{aligned}
$$

$$\tanh(0.667583) = 0.5834$$

Of course, with initial weights being random, the predicted outputs are not expected to be close to the desired (or "target") outputs, $o_1 = 0.08$ and $o_2 = 0.83$. The predicted outputs $o_1 = 0.3860$ and $o_2 = 0.5834$ are effectively random.

Let's calculate just how "wrong" the predicted outputs are. We do this using a loss function. We will use the squared error function:

$$E = \frac{1}{2} \sum (target - predicted)^2$$

The squared error loss function does just what it says on the tin: it finds the difference between the target and predicted value (the "error") and squares it. We then sum all of the errors.

You may note that there is a $\frac{1}{2}$ constant in the function. As long as we scale by a constant (and consistently do so whenever we use the loss function), this will not affect our calculations. As long as every instance of the function uses the same constant they will still be comparable with one another. The reason we include this $\frac{1}{2}$ constant is to make it easier when we differentiate later on, to cancel out the power of two later on.

The error for o_1 is:

$$
\begin{aligned}
Eo_1 &= \frac{1}{2}(0.08 - 0.3860)^2 \\
&= \frac{1}{2}(-0.306)^2 \\
&= \frac{1}{2}(-0.306)^2 \\
&= \frac{1}{2}(0.093636) \\
&= 0.046818
\end{aligned}
$$

The error for o_2 is (simplified):

$$\begin{aligned} Eo_2 &= \frac{1}{2}(0.83 - 0.5834)^2 \\ &= 0.03040578 \end{aligned}$$

The total E is:

$$\begin{aligned} E &= 0.046818 + 0.03040578 \\ &= 0.07722378 \end{aligned}$$

Now comes the actual backpropagation algorithm: we would like to find how much each output connection contributed to the final error, and tweak the weights on each connection to reduce the error.

How do we find how much each connection (or each weight) contributed to the error? We want to know how much a change in some connection weight w, affects the total error, E. Expressing the problem in this way makes it obvious that the solution to our problem is by using calculus: differentiation. We can write the problem as such:

$$\frac{\partial E}{\partial w}$$

Clearly we must work backwards from the output layer towards the input layer, because the first step (with the information we have), must be to calculate how much the final layer of connections contributed the the final error before we can worry about any previous layers.

We do not have enough information yet to calculate , so let's start with what we know. We can differentiate E with respect to o_1:

$$E = \frac{1}{2}\left[(target_{o1} - predicted_{01})^2 + (target_{o2} - predicted_{o2})^2\right]$$

$$\begin{aligned} \frac{\partial E}{\partial o1} &= -1 \cdot 2 \cdot \frac{1}{2}(target_{o1} - predicted_{o1})^{2-1} + 0 \\ &= -1 \cdot (target_{o1} - predicted_{o1}) \\ &= predicted_{o1} - target_{o1} \end{aligned}$$

(In the second line, the -1 coefficient multiplier is due to the chain rule; and the second term is a constant with respect to $o1$ so it differentiates to zero.)

We know the predicted and target values for $o1$! This means we can calculate :

$$\begin{aligned}\frac{\partial E}{\partial o1} &= predicted_{o1} - target_{o1} \\ &= 0.3860 - 0.08 \\ &= 0.306\end{aligned}$$

Next, we calculate how much the output of $o1$ changes with respect to its total input, n. Effectively, the only difference between each TLU's total input and its output is the activation function. In this case, n is the input to the TLU's input before the activation function was applied. You may go back and double-check this for yourself. In our case, we must find the derivative of tanh, which is known to be sech^2 .

$$\begin{aligned}o1 &= \tanh(n) \\ \frac{\partial o1}{\partial n} &= \text{sech}^2(0.407057) \\ &= 0.85103\end{aligned}$$

Now we finally get to w, the connection whose weight we are interested by; we have an equation for n using w, so we can find the derivative: the basic equation for finding the weighted sum of inputs.

Below, the weight w is for the connection h_1o_1, the weight wx is for the connection h_1o_2 (we are not interested by wx just yet, and we know the term will 'disappear' to zero when we differentiate with respect to w). The h_1 refers to the output of the TLU 'hidden 1':

$$\eta = (w \cdot h_1) + (wx \cdot h_2)$$

(If you are confused about where this equation comes from, refer to earlier in the paper where we calculate the output from each TLU. We've just changed the notation and variable names to apply to the layers we are looking at.)

When we find the derivative, the w coefficient differentiates to 1 and the $(wx \cdot h_1)$ differentiates to zero, so we are left with:

$$\begin{aligned}\frac{\partial n}{\partial w} &= h_1 \\ &= 0.6482\end{aligned}$$

This is all great, but how do we put it all together to find $\frac{\partial E}{\partial w}$, like we wanted? Well, thanks to the chain rule, it is possible to express $\frac{\partial E}{\partial w}$ as:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial o1} \cdot \frac{\partial o1}{\partial n} \cdot \frac{\partial n}{\partial w}$$

This uses all of the derivatives we have already calculated! It's now trivial to put them all together:

$$\begin{aligned} \frac{\partial E}{\partial w} &= 0.306 \cdot 0.85103 \cdot 0.6482 \\ &= 0.1688 \end{aligned}$$

To tweak the weight w, we subtract this value from the current weight for that connection—in this case, the current weight is 0.46. We should actually multiply this value by the learning rate, n, of our Multilayer Perceptron, $0.46 - 0.1688n$. To keep things simple, however, we will just use a learning rate of 1.

Therefore, our tweaked weight for the h_1o_1 connection would be 0.2912.

However, we do not tweak each weight as we go along, because we still must calculate $\frac{\partial E}{\partial w}$ for the other connections in the final layer. For the sake of brevity, I won't write out each step. Feel free to try this step for yourself to see if you achieve the same result:

Connection	$h_1o_2 = -0.105444$
Connection	$h_2o_1 = 0.189040$
Connection	$h_2o_2 = -0.118080$

We now have the values we will use to update the weights on the output layer. We use a similar process for the hidden layer. However, this time the output of the TLUs h_1 and h_2 affects both o_1 and o_2. This means it contributes to the error of both TLUs in the output layer. This is easily accounted for:

$$\frac{\partial E}{\partial h_1} = \frac{\partial Eo_1}{\partial h_1} + \frac{\partial Eo_2}{h_1}$$

(h_1 refers to the output of the TLU 'hidden 1'.)

Using the chain rule on the first term (just as in the previous step, n refers to the input to o_1):

$$\frac{\partial Eo_1}{\partial h_1} = \frac{\partial Eo_1}{\partial n} \cdot \frac{\partial n}{\partial h_1}$$

We can't derive the $\frac{\partial Eo_1}{\partial n}$ immediately. However, we did find the derivatives $\frac{\partial Eo_1}{\partial o_1}$ and $\frac{\partial o_1}{\partial n}$ in the output layer step (you can go back and see for yourself!) This is all we need! Using the chain rule, we can find $\frac{\partial Eo_1}{\partial n}$ by:

$$\frac{\partial Eo_1}{\partial n} = \frac{\partial Eo_1}{\partial o_1} \cdot \frac{\partial o_1}{\partial n}$$

Since we have the needed values:

$$\begin{aligned}\frac{\partial Eo_1}{\partial n} &= 0.306 \cdot 0.85103 \\ &= 0.260415\end{aligned}$$

Now let's deal with the $\frac{\partial n}{\partial h_1}$, shall we? We already have an equation for n that uses the output of h_1: the equation for finding the weighted sum of inputs. The instance of the equation we are using (as we may be able to infer from the context we are using it in) is the connection h_1o_1. We found its derivative with respect to w (the weight of the connection h_1o_1) when we were dealing with the output layer; now we are finding its derivative with respect to the output of h_1:

$$n = (w \cdot h1) + (wx \cdot h2)$$

$$\begin{aligned}\frac{\partial n}{\partial h_1} &= w \\ &= 0.46\end{aligned}$$

We can now find $\frac{\partial Eo_1}{\partial h_1}$:

$$\begin{aligned}\frac{\partial Eo_1}{\partial h_1} &= \frac{\partial Eo_1}{\partial n} \cdot \frac{\partial n}{\partial h_1} \\ &= 0.260415 \cdot 0.46 \\ &= 0.11979\end{aligned}$$

If we repeat these few simple steps for the other term needed to calculate $\frac{\partial E}{\partial h_1}$, the equivalent steps for o_2:

$$\frac{\partial Eo_2}{\partial h_1} = \frac{\partial Eo_2}{\partial n} \cdot \frac{\partial n}{\partial h_1}$$

$$\frac{\partial Eo_2}{\partial n} = \frac{\partial Eo_2}{\partial o_2} \cdot \frac{\partial o_2}{\partial n}$$

$$\begin{aligned} &= -0.2466 \cdot 0.65965875 \\ &= -0.162672 \end{aligned}$$

$$\begin{aligned} \frac{\partial n}{\partial h_1} &= w \quad \text{(this time for the connection } h_1 o_2\text{)} \\ &= 0.19 \end{aligned}$$

$$\begin{aligned} \frac{\partial Eo_2}{\partial h_1} &= \frac{\partial Eo_1}{\partial n} \cdot \frac{\partial n}{\partial h_1} \\ &= -0.162672 \cdot 0.19 \\ &= -0.030908 \end{aligned}$$

Finally:

$$\frac{\partial E}{\partial h_1} = \frac{partial Eo_1}{\partial h_1} + \frac{\partial Eo_2}{\partial h_1}$$

$$\begin{aligned} &= 0.11979 - 0.030908 \\ &= 0.088882 \end{aligned}$$

Now we have accounted for the neuron's effect on multiple outputs, we can continue with the rest of the steps as we did with the output layer:

$$\begin{aligned}\frac{\partial h_1}{\partial n_{h1}} &= \text{sech}^2(n) \\ &= \text{sech}^2(0.7724) \\ &= 0.579677\end{aligned}$$

$$\begin{aligned}\frac{\partial n_{h1}}{\partial w_{i1,h1}} &= i_1 \\ &= 0.56\end{aligned}$$

$$\begin{aligned}\frac{\partial E}{\partial w_{i1,h1}} &= 0.088882 \cdot 0.579677 \cdot 0.56 \\ &= 0.0288528\end{aligned}$$

And we calculate for the other connections in the hidden layer. Once again, for the sake of brevity, I won't write out each step (this is probably going to take a while for me to compute!):

Connection	$i_1h_2 = 0.0452777$
Connection	$i_2h_1 = 0.0123655$
Connection	$i_2h_2 = 0.0194047$

Adjust all the weights in the Multilayer Perceptron:

Connection	$i_1h_1 = 0.8289$
Connection	$i_1h_2 = 0.8653$
Connection	$i_2h_1 = 0.07237$
Connection	$i_2h_2 = 0.9394$
Connection	$h_1o_1 = 0.2912$
Connection	$h_1o_2 = 0.08456$
Connection	$h_2o_1 = 0.3390$
Connection	$h_2o_2 = 0.6319$

Testing the MLP with the new weights:

$$\begin{aligned}h_1 &= (0.8289 \cdot 0.56) + (0.07237 \cdot 0.24) + (0.31 \cdot 1) \\ &= 0.791553\end{aligned}$$

$$\tanh(0.791553) = 0.6592877942$$

$$\begin{aligned} h_2 &= (0.8653 \cdot 0.56) + (0.9394 \cdot 0.24) + (0.24 \cdot 1) \\ &= 0.950024 \end{aligned}$$

$$\tanh(0.950024) = 0.7397939164$$

$$\begin{aligned} o_1 &= (0.2912 \cdot 0.6592877942) + (0.3390 \cdot 0.7397939164) \\ &= 0.442775 \end{aligned}$$

$$\tanh(0.442775) = 0.415942$$

$$\begin{aligned} o_2 &= (0.08456 \cdot 0.6592877942) + (0.6319 \cdot 0.7397939164) \\ &= 0.523225 \end{aligned}$$

$$\tanh(0.523225) = 0.480185$$

$$E = 0.22916$$

Unfortunately (and, I admit, somewhat depressingly considering the time and effort this took) our error from the loss function worsened after one pass of the backpropagation training algorithm. Other than human error from manually computing each step—which is possible with such a large problem as this—it is not uncommon for the algorithm to make the error smaller with every single interaction. This is because backpropagation is a form of gradient optimisation, and such algorithms do not guarantee a smaller error at every single step.

Furthermore, we opted to arbitrarily set the learning rate to $\eta = 1$ for simplicity's sake. It is likely that we could have seen a better result with a more thoughtful selection for η.

Over time, if we were to perform many many iterations and passes of this algorithm, we would ultimately expect to see the prediction converge with smaller error each time.

4 Key Points and Conclusion

The backpropagation algorithm runs through the MLP neural network and makes its first predictions (forward pass). From these predictions, a loss function

calculates the error.

It then finds how much each weight in the network contributed to the final error by working backwards and using partial derivatives. Knowing this, the algorithm duly tweaks the connection weights in the network to hopefully reduce the error. This process is repeated many times over, with each pass being known as an 'epoch.'

Although it cannot be demonstrated in a walkthrough of a single pass, as we have done here, the instances that are input at the beginning of the network will be altered with each epoch, to train the model on the entire training set.

Each instance will pass through the network multiple times during training. In our example, our input instances were 0.56 and 0.24 but in practice we would use a training set with many instances. In reducing the error with each pass, the backpropagation algorithm will produce weights that eventually converge on being able to be a representative predictor of the whole set.

However, as we have seen firsthand, each epoch individually may not reduce the error.

5 Questions

(For the sake of the reader, and also for myself should I revisit my learning):

1. Why is it important that the activation function is a continuous function? Why did we choose tanh in particular, and what is the alternative, more common, activation function?
2. We want to know how much a change in some connection weight w, affects the total error, E. How can we express this problem with mathematical notation so we can go about solving it with calculus?
3. What extra step in our method do we have to consider when finding the contribution of the hidden layer connection weights to the total error?

6 References

Matt Mazur attempted a similar walkthrough of the backpropagation algorithm, to admittedly more success (in terms of actually reducing the network's error). His MLP was set up differently to mine—most notably, he used two bias neurons—and he opted for the more common sigmoid activation function. The calculus may be harder to follow, as he spent less time breaking down each step:

- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

My first port of call for understanding MLPs, TLUs and the backpropagation algorithm was of course our textbook:

- Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (iBooks)