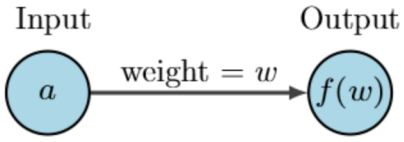
Backpropagation

It is time to teach the network how to learn. So far, we have used the Multi-Layered Perceptron as a forward propagating network. Currently, our model is not good at classifying the images since the weights and biases are randomized. On the other hand, with the smaller networks (see pg. 21, Fig. 9), it was possible to determine which set of weights and biases produced the desired output. However, the MLP model we have created for recognizing handwritten digits has 11,935 weights and biases. So it would not be feasible for someone to figure out which set of 11,935 weights and biases make the network produce the correct prediction for the 70,000 input images. Instead, we will implement the backpropagation algorithm, which will allow the network to determine the weights and biases that produce the best predictions for each input image.

Implementing the backpropagation algorithm in TensorFlow is surprisingly easy. The API abstracts all of the heavy lifting that the algorithm computes to determine the correct set of weights and biases. It turns out that all the algorithm is doing behind the scenes is just a ton of calculus. Notably, the algorithm implements a process known as gradient descent. To see how this works, let us step away from our MLP and dive into the calculus with some examples.

The Simple Network

Instead of thinking about images and the relevant matrices and tensors, let us simplify the network to build back up to our MLP. To start, the simplest network we can create is one that takes in a single input and produces a single output.



To compute our network's output $f(w)$, we multiply the input by the weight to get $f(w) = aw$. Again, the goal of this network is to produce the desired output, call it $\hat{y}$, such that our actual output $f(w) = \hat{y}$. However, for a fixed input a, we are not guaranteed that $f(w)$ will equal the desired output. We could easily figure out w

for a given a using algebra with this simple network. But since we will not be able to do that with our MLP, we should pretend that not an option. This leaves us to implement the backpropagation algorithm.

Consider the likely case when the output $f(w)$ is not equal to $\hat{y}$. If we cannot use algebra, we must estimate how much we need to change w so that $f(w) = \hat{y}$. To find this estimate, we will use a process known as gradient descent. To explore this algorithm, let us look at an example by giving these variables some values and solving for w.

$$a = 2, w = 0.8, \hat{y} = 1$$

$$f(w) = 2(0.8) = 1.6$$

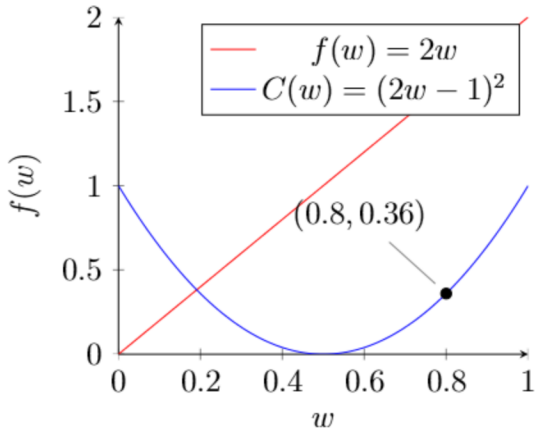
For the fixed input $a = 2$, this network outputs 1.6, but we want $f(w) = 1$. The goal is to figure out what value for w will output 1. We will start by considering the squared distance between the actual output $f(w)$ and the desired output $\hat{y}$:

$$C(w) = (f(w) - \hat{y})^2.$$

This distance function $C(w)$ is known as the cost or error function, which tells us how bad our output is. If $f(w)$ is close to $\hat{y}$, the cost function will be closer to zero, resulting in a better output. So, let us find out how close the output of our network is to the desired output:

$$C(w) = (1.6 - 1)^2 = 0.36.$$

As shown, the network outputs something close to $\hat{y}$, but we can do better. We will start by examining the functions $f(w) = 2w$, $C(w) = (f(w) - \hat{y})^2$ and plot them.



If we want the output of the network $f(w)$ to equal $\hat{y}$, then we must find the local minimum of the cost function $C(w) = (2w-1)^2$. We can see the local minimum is 0.5, and to get there, we need to make small changes to w so that $C(w)$ trends down its parabola arriving at the local minimum. To make these small changes, we must incrementally update the current weight in the network to eventually arrive at the minimum such that $C(w) = 0$, for a fixed a. To determine how much we need to change w, we first need to know how the output $f(w)$ affects the cost and then how the weight affects the output. We can do this by taking the partial derivative of the cost function $C(w)$ with respect to the weight w. In this example, we take the partial derivative because we only want the rate of change of w where the other variables are treated as constants. This partial derivative $\frac{\partial C}{\partial w}$ is the slope of the cost function relative to the weight.

To compute the partial derivative of the cost function, we use the chain rule of derivation to split the derivative into parts.

$$\frac{\partial C}{\partial w} = \frac{\partial f(w)}{\partial w}\frac{\partial C}{\partial f(w)} = 2a(f(w) - \hat{y})$$

Thus, $\frac{\partial C}{\partial w}$ tells us how much we need to change our weight, so if we subtract the partial derivative from our original weight, we should get closer to the minimum. With this naive approach, we can calculate the new weight w' as follows,

$$w' = w - \frac{\partial C}{\partial w} = w - 2a(f(w) - \hat{y})$$

With this equation, let us update the starting weight $w = 0.8$ from our simple network for the fixed input $a = 2$, and repeat the process multiple times to minimize the cost function.

c	w	w′	f(w′)
1	N/A	0.8	1.6
2	0.8	-1.6	-3.2
3	-1.6	15.2	30.4
4	15.2	-102.4	-204.8
5	-102.4	720.8	1441.6
6	720.8	-5041.6	-10083.2

Well that's not getting us closer to $f(w) = 1$. It is getting worse; what happened? Well, our updates to w were just too large. If we were to keep going with weight updates, it would tend toward positive and negative infinity. In contrast, if we make smaller changes to the weight w, still using the derivative of the cost function, we will not bounce between positive and negative infinity. We take a small portion of the partial derivative $\frac{\partial C}{\partial w}$ by multiplying it by some fraction r. This value is known as the learning rate, and this is usually a chosen value before performing backpropagation. With this in mind, let us recalculate the new weights, where $r = 0.1$.

$$w' = w - r * \frac{\partial C}{\partial w} = w - 2ar(f(w) - \hat{y})$$

We can look at few weight updates again to see the new output of the network.

c	w	w′	f(w′)
1	N/A	0.8	1.6
2	0.8	0.56	1.12
3	0.56	0.512	1.024
4	0.512	0.5024	1.0048
5	0.5024	0.50048	1.00096
...	...	...	...
24	0.5000000000000007	0.5000000000000001	1.0000000000000002
25	0.5000000000000001	0.5	1.0

That's better, and after 25 weight updates our network has been optimized such that $f(w) = \hat{y}$.

This is the first part of the backpropagation algorithm: gradient descent! We can easily implement these weight updates in TF, by defining our network f, the cost function C and its partial derivative using Tensors as follows.

SIMPLE NETWORK

```
a = tf.Variable(2.0)
w = tf.Variable(0.8, trainable=True)
d = tf.Variable(1.0)

def f(a, w):
    return a * w

def cost(actual_y, desired_y):
    return tf.math.square(actual_y - desired_y)

def cost_partial():
    return 2*a*(f(a,w) - d)
```

Now let us define a function for updating the weights using our defined partial derivative of the cost function.

SIMPLE NETWORK

```
def update_weights(updates, lr=1.0):
    for i in range(updates):
        dw = cost_partial()
        w.assign_sub(lr * dw)
```

This function takes two arguments: the number of weight updates we want to perform and a learning rate (which is set to 1 by default). Then for each weight update, we call *cost_partial()* which calculates $\frac{\partial C}{\partial w}$. To update the weight, we use the function *assign_sub()* to subtract the partial derivative times the learning rate from the original weight in place. We can only do this with **Variables** since they are mutable data types. This way, we do not have to return the new weight since its

update is applied to the global variable w. Now, we can perform the weight updates with and without the learning rate.

NO LEARNING RATE

```
update_weights(5)
print("w': ", w, " f(w'): ", f(a,w))
```

WITH LEARNING RATE

```
update_weights(25, lr=0.1)
print("w': ", w, " f(w'): ", f(a,w))
```

After running these two functions, we would find that their outputs match the last line from our tables we manually calculated from before. As we continue through the examples of the backpropagation algorithm, we will be calculating more and more partial derivatives. It would be possible to define those partials as a function in TF as we did here. However, this is not a good long-term solution, especially if we want to find the partial derivatives of a large and complicated function. This is why TensorFlow has a module call *GradientTape()* which performs automatic symbolic differentiation. It is used to calculate the derivatives of any function defined within it or with respect to a trainable **Variable**. It calculates the derivatives by recording the defined operations to the *tape*, which is stored in memory. Then these calculations can be used to find the partial derivatives with respect to any well-defined Tensor. So, let us use this module to find our partial derivative with respect to the weight, $\frac{\partial C}{\partial w}$.

AUTO DIFFERENTIATION

```
def tf_partial():
    with tf.GradientTape() as tape:
        c = cost(f(a, w), d)
        dw = tape.gradient(c, [w])
    return dw
```

To perform the automatic differentiation, the *tape* needs to record the operations during the forward pass of the function. In our case, this would be calculating the

output of our function $f(a, w)$, and then plugging it into the cost function *cost()* with our desired output. TensorFlow will only record these operations if they are defined within the **with** block or manually tracked. Using the former, we calculate cost C by calling each function to compute the cost function's actual value. Now that these operations are recorded on the tape (in the background) in memory, we can then calculate $\frac{\partial C}{\partial w}$, by calling the function *tape.gradient()* which will find the value of the partial derivative. This function takes two arguments, the value that the operations defined and a list of the trainable variables we want to take the partial derivatives with respect to. This is why we originally defined w in the code with the trainable attribute set to **True**. We cannot call *tape.gradient()* on any non-trainable variables. If correctly input, the function will return all of the defined partial derivatives, and here it just returns the single value of $\frac{\partial C}{\partial w}$ which we called dw. We have our partial derivative dw, and we can return it to be used in the weight updates.

It is important to note that all of the resources allocated to the *tape* are automatically deallocated at the end of the **with** block. Since the resources only need to last until they are used for the differentiation, they are immediately released to avoid holding on to garbage information. Furthermore, any TF **Variable** defined in a **Module** are automatically set as trainable, meaning that the weights and biases in our MLP will be automatically tracked and ready to be differentiated. With this new function, let us add this new method into our weight update function. To ensure that *tf_partial()* is calculating the same value, let us call both functions and then assert they are the same.

NEW WEIGHT UPDATE

```
def update_weights(updates, lr=1.0):
    for i in range(updates):
        dw = cost_partial()
        dw2 = tf_partial()
        assert(dw == dw2)
        w.assign_sub(lr * dw)
```

If we run the code shown on page 42, we should get no errors and the same outputs. The code shows that the partial derivative calculated by TF is the same as

the one we found by hand. The function *tape.gradient()* is highly optimized using symbolic differentiation in coordination with the computational graph to ensure the most efficient calculations of partial derivatives. It is out of the scope of this paper to explain the algorithm that finds these partial derivatives. Thus, now that we have a way to calculate partial derivatives in TF, we will use this on our MLP once we have gone through the entire backpropagation algorithm.

It turns out for a fully connected model similar to the MLP, this gradient descent algorithm we just implemented is essentially the same. The only thing that changes is the complexity of the output function. This depends on how many layers there are in the network and what functions are used. Now that we know how to update the weights for a single weight let us look at how we will update the weights for multiple layers.