Understanding Neural Networks with TensorFlow

David Pojunas

Hendrix College

A thesis submitted for the degree of

Bachelor of Arts in Computer Science and Mathematics

Hendrix College 2021

Abstract

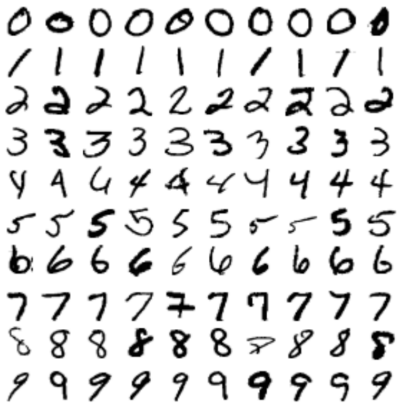
Neural Networks are a class of machine learning models, inspired by how the brain operates. There are many different types of neural networks with multiple use cases and levels of complexity. Without prior knowledge of these models, it can be an intimidating subject to learn.

The goal of this paper is to understand the fundamental concepts behind different neural network architectures. We use TensorFlow to understand these concepts by implementing a Multi-Layer Perceptron (MLP). The abstractions provided by TensorFlow helps us translate the design of the MLP to other networks architectures. In the end, we build an MLP to recognize handwritten digits!

Contents

Introduction

A classic problem in neural networks is trying to recognize handwritten digits. Looking at the images below, we can effortlessly recognize all 100 unique handwritten digits. Our brain processes these images and interprets what they represent with ease. However, the simplicity of this process is deceiving. Once our eyes focus on the image, the information goes to the cerebral cortex that processes the visual information. The cerebral cortex contains five cortices that allow us to do very complex image processing. Essentially, we have a supercomputer for processing visual information powered by the billions of connections between the millions of neurons in our brain.


(a) Good samples

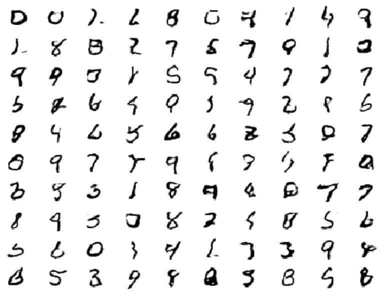

(b) Bad samples

Figure 1: MNIST Images

Even with more poorly written digits, we can still instantly infer what the digit should be. We can do this because our brains are very good at recognizing patterns. From a young age, we have all had lots of practice reading and writing letters and digits. However, we had to learn what these digits are by understanding their design and meaning at some point. We have taught our brains how to recognize these digits through training and repetition. Moreover, when we encounter more poorly written digits, we can compare their shape to the digits we already know.

Neural networks are a subclass of machine learning algorithms that take inspiration from our brain's neural connections and learning models. They can approach this problem by taking many input images known as training samples to develop an optimized system for recognizing handwritten digits. The more experience (or training samples) a neural network has seen, the better it will be at recognizing any digit it sees in the future.

In this paper, we will learn how to build a neural network to recognize handwritten digits. Additionally, we will use TensorFlow, a machine learning API, to build the neural network from scratch. This API will give us a deeper understanding of how these networks' designs change to solve a broader range of problems.

In the field of machine learning, there are two predominant models for learning: supervised and unsupervised. Supervised learning attempts to solve classification problems, whereas unsupervised learning is used the recognize patterns in data. The goal of unsupervised learning is to learn these patterns without using labels or guides. On the other hand, with supervised learning, the labels are used explicitly in the training process, where the goal is to find a relationship in the input data allowing us to produce the desired output.

We will use supervised learning to teach a neural network how to recognize handwritten digits. We can think about this supervised learning as a mapping between a set of inputs to a set of discrete outputs. For our problem, the neural network will map the input images to a prediction of its label (digit). Our neural network will use calculus in the training process to optimize this mapping. We will see how calculus breaks down this problem into a relatively simple minimization problem using partial derivatives. TensorFlow will help bridge the gap between the networks' discrete mapping and structure to calculus's continuous nature. In the end, we will gain an understanding of how the design of the TensorFlow API can be leveraged to build more complex neural networks that extend beyond learning to recognize handwritten digits.

What is TensorFlow?

TensorFlow (TF) is a powerful machine learning API with an extensive ecosystem of tools and libraries for developing neural networks. The core API provides clear abstractions for building and training neural networks. It also contains essential data analysis tools for testing and improving the performance of networks. These core libraries are broken apart by the more fundamental concepts behind all neural networks. In general, the TF framework has two main components: building a network and running a network. By itself, a neural network is just a graph of nodes and edges. It's only when data is passed through the network that it does something interesting. This means we have to build the graph before anything else.

TensorFlow represents neural networks as computational graphs. They are used to abstract and compute mathematical functions. For example, we can create a computational graph with the following equation.

$$f(A, B, C) = (A * B) - (B + C)$$

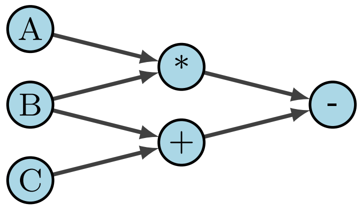


Figure 2

This powerful idea separates the data in a network from its operations, allowing TF to execute those operations in parallel. In the API there is a simple way to tell TF which operations to offload to a GPU or another CPU. We will use this to improve the performance of our networks. But for now, it will just be our motivation for the potential power behind using a computational graph.

We can think of coding a program in TF as creating this computational graph. However, we do not have to create this graph directly. *Eager execution* is a programming environment that immediately evaluates operations (in TF 2.0 this is on by default). In this mode, we don't have to build the graph first and then run the operations later. We can code naturally in Python data structures and use Python control flow. Once the Python code is executed, the TF interpreter will infer the computational graph dynamically. This does not mean we can completely forget about the computational graph underlying our Python code. As we build networks in Python, TF will try and optimize the graph once we run data through the network. So it's important to understand how we can write our code while still being mindful of this unseen graph. Most of our potential errors will be caused by how the API structures its classes, objects, and functions. This will make our code more robust and scalable as we make more complex networks.

With this in mind, to create our Multi-Layer Perceptron, we need two things from the API: Tensors and Modules.

A Tensor is a multi-dimensional array with a uniform type. It stores information with two properties: it has one type (float32, int, or string, etc) and a shape (i.e. what are the dimensions of the array).

A Module is a container to store Tensors, other Modules, and operations. Modules will be useful to subclass our Python classes. This way we can use the built-in functions of the computational graph flow from the perspective of a Python class.

Perceptron

A Perceptron is a single layer neural network biologically inspired by the neurons that send and receive signals in the brain. Neurons receive signals through root-like extensions called dendrites. Those signals are then modulated by the neuron's electric potential. A neuron fires when the signal is strong enough to be sent through the axon. If fired, the output signal is passed onto many more neurons, as demonstrated in the picture below.

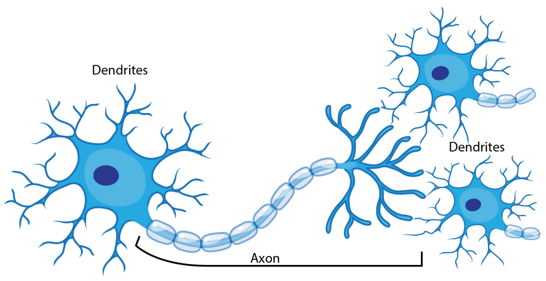


Figure 3: Image borrowed from ASU Ask A Biologist [1]

A Perceptron is an artificial neuron represented as a mathematical model, where the dendrites are numerical inputs and the modulations are weights. The artificial neuron fires if the weighted sum of the inputs reaches some threshold. In general, a Perceptron takes in a set of inputs and produces a single output. With this paradigm, we can create complicated behaviors from a simple network. Below is a diagram of a Perceptron with two inputs.

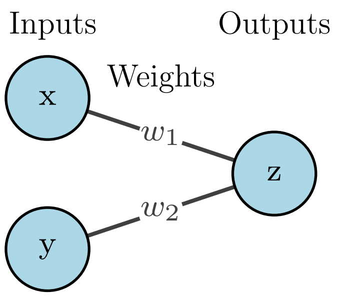


Figure 4

The output is computed by taking the sum of the inputs multiplied by their associated weights. This operation is defined below for two arbitrary inputs and weights.

$$x * w_1 + y * w_2 = z$$

The weights (w_1, w_2) change how much each input (x, y) influences the output of the Perceptron. When given a set of inputs, the weight value will change the behavior of the network. An activation function decides whether or not the Perceptron should fire. This activation function linearly separates the output values based on a threshold. An output above the threshold tells the Perceptron to "fire" and send a value of one. Otherwise, when the output falls below the threshold, the Perceptron sends a value of zero. This output allows us to classify the inputs based on the output of the network. But before we define this activation function we need to implement the Perceptrons weighted sum in TF. So for now, we'll think about output of the Perceptron as the equation $x * w_1 + y * w_2 = z$.

Matrix Multiplication

Since TF stores its values as Tensors, let us rewrite our equation in terms of matrices, where the dot product is the output.

$$\begin{bmatrix} x & y \end{bmatrix} * \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} x * w_1 + y * w_2 \end{bmatrix} = [z]$$

Naturally, TensorFlow has the dot product and matrix multiplication built-in for Tensors. As a refresher, let us compute the matrix multiplication of a (2×2) matrix by a (2×1) matrix where the first value is the number of rows and the second is the number of columns.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 10 \\ 20 \end{bmatrix} = \begin{bmatrix} 10 + 20 \\ 60 + 80 \end{bmatrix} = \begin{bmatrix} 30 \\ 140 \end{bmatrix}$$

Again, we use the dot product to multiply the columns of the (2×1) matrix by each row in the (2×2) matrix and then take the sum.

We can generalize matrix multiplication by considering matrix A with dimensions $(m \times n)$ and matrix B with dimensions $(n \times p)$. To multiply two matrices $A * B$, we have to ensure that A has the same number of columns as there are rows in B. This is true of A and B so the resulting matrix of their product has the following dimensions.

$$\dim(A * B) = (m \times n) * (n \times p) = (m \times p)$$

Note that $A * B \neq B * A$. We can't guarantee that we would even be able to multiply $B * A$, since we do not know that B has the same number of columns p as there are m rows of A. We can only multiply $B * A$ if $p = m$. When this is true, $B * A \neq A * B$ and the resulting matrix will have the following dimensions.

$$\dim(B * A) = (n \times p) * (p \times n) = (n \times n)$$

Therefore, we can only multiply two matrices if they satisfy the row-column condition. Note, we will always know the resulting dimension of the multiplication without having to compute the result. The reason why this matters is that TF implements this matrix multiplication and knowing these operations will cause a lot less headache over multiplying and manipulating Tensors. Now let us look at how we can use this multiplication to compute the output of the Perceptron from before.

Tensor Multiplication

To start coding in TF, we first need to import the API at the top of our Python file (see Appx. pg. 80 for details on installation). We will show this here, but any code after will not include this line.

Below is the code for the Perceptron. We have assigned two arbitrary values for the inputs and weights each with their own role in computing the output of the Perceptron.

Code

```
import tensorflow as tf

inputs = tf.constant([[1.0,0.0]])
weights = tf.Variable([[0.5],[0.5]])
outputs = tf.matmul(inputs, weights)
```

Printed Outputs

```
inputs:  tf.Tensor([[1. 0.]], shape=(1, 2), dtype=float32)
weights:  <tf.Variable 'Variable:0' shape=(2, 1) dtype=float32,
            numpy= array([[0.5],[0.5]], dtype=float32)>
outputs:  tf.Tensor([[0.5]], shape=(1, 1), dtype=float32)
```

It's important to note that our printed statements might differ slightly, which can depend on whether we use Python lists or NumPy arrays. TensorFlow will happily use either version without changing anything other than the print statement. Examining this code, we have taken our matrix interpretation of the Perceptron and translated it directly to Tensors.

The inputs are a **tf.constant** because inputs to a Perceptron will never change. The weights are a **tf.Variable** because these will need to change in the training process once we use this in a Multi-Layer Perceptron. The outputs are a generic Tensor since they are defined by an operation. In other words, we have created two data nodes in the computational graph that tell TF how to compute the outputs using matrix multiplication. These building blocks are defined in TF with the following properties.

A **tf.constant** is an immutable Tensor. Once referenced by a Python variable its values cannot be changed. It can be reshaped which changes how the array is indexed. In memory, a Tensor will always store its values in a one-dimensional array. A **tf.Variable** is a mutable Tensor that cannot be reshaped. Unlike the constant, its values can be changed. Matrix operations can be applied on any kind of Tensor and its output will be a new Tensor.

We can see from the printed values that **tf.matmul()** uses the dot product to multiply the two Tensors. The shape of the input (1,2) by the shape of the weights

(2,1) defines the shape of the output (1,2) * (2,1) = (1,1). However, we have to be careful when using these operations, and the order of the parameters matters. If we swap the parameters for **tf.matmul()**, we a completely different Tensor.

```
Swapped Inputs
outputs = tf.matmul(weights, inputs)


outputs:  tf.Tensor(
[[0.5 0. ]
 [0.5 0. ]], shape=(2, 2), dtype=float32)
```

With the swapped inputs, we are multiplying the weights (2,1) by the inputs (1,2) for an output with shape (2,1) * (1,2) = (2,2). Here's a perfect example when $A * B \neq B * A$. This might seem simple, but as we add more dimensions to our Tensors we can use this insight to easily perform operations on any size Tensors.

Activation Function

Now that we have a way to represent our Perceptron through Tensor operations, we can go back and define the activation function. Let uss move away from TF for a moment. Up until now, we've represented the output as the matrix multiplication of the weights w by the inputs a, where the output is $w * a = y$.

$$a = \begin{bmatrix} a_1 & a_2 \end{bmatrix}, w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, y = [y]$$

Without an activation function, the output y could be any value between positive and negative infinity. The activation function sets bounds on the output, which signals to the Perceptron when to fire. In our case, we want the Perceptron to output a one (fired) or a zero (not fired). To do this classification, we linearly separate the output values by some threshold value C. We will write this activation function as $\delta_C(y)$, where,

$$\delta_C(y) = \begin{cases} 1 & \text{if } w * a > C \\ 0 & \text{if } w * a \leq C \end{cases}$$

However, this exact definition of the activation function is a bit clunky to implement when creating networks of Perceptrons. Relying on a non-fixed value for the output forces us to make unnecessary calculations. So, we can subtract the threshold C from both sides to get the new activation function $\delta_C(y)$.

$$\delta_C(y) = \begin{cases} 1 & \text{if } w * a - C > 0 \\ 0 & \text{if } w * a - C \leq 0 \end{cases}$$

By doing this, we have turned the threshold value into what's known as a bias. The bias allows us to shift the output of the Perceptron to the left or the right for different classifications. Notice that $\delta_C(y)$ is a step function. With this change, the Perceptron now depends on the values of the weights, the bias, and this activation function.

Creating the Perceptron

Now that we have all pieces of a fully-fledged Perceptron. It turns out that we can do a lot of different types of classification problems. We are going to use this model of the Perceptron to compute AND, OR gates. When completed, we should be able to produce the tables of their logic.

AND		
a_1	a_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR		
a_1	a_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Let us jump back to TF and add in the bias and activation function. However, our original Perceptron was a pretty exposed data structure, so we will redefine the Perceptron as a Python class, sub-classed by **tf.Module**. By sub-classing it, we are telling TF that the objects inside the Python class should be one node in the computational graph. This means any Python reference to our Perceptron class will always have the same structure and compute the same operation. Below is the code for our new and improved Perceptron.

Perceptron
```python
class Perceptron(tf.Module):
    def __init__(self, weights, biases, name=None):
        super().__init__(name=name)
        self.weights = tf.Variable(weights, name="weights")
        self.biases = tf.Variable(biases, name='biases')
    def __call__(self, x):
        return tf.matmul(x, self.weights) + self.biases
```

A lot is going on here, so we'll break it down, line by line.

First, we define our class **Perceptron** sub-classed by **tf.Module**. Then we override the inherited constructor from **tf.Module** with our constructor and parameters. We call the constructor of **super()** to inherit all the methods and properties from **tf.Module**. Then we can use the optional parameter *name*, which is useful for labeling objects inside the computational graph. In our constructor, we make the parameter optional as well by setting the default to **None**.

Then we define the data inside **Perceptron** which are the weights and biases where their **tf.Variable**'s take any array defined by the parameters. We also give them labels in the same way we do with the Perceptron.

Finally, we override the Python *call* constructor, which makes it possible to use the instance variable of a class as a function, which is how the inputs will be given to our Perceptron. In other words, this *call* method is the operation that the Perceptron uses to compute its outputs.

Yet, we are still missing the activation function which we said was a step function. However, there are other activation functions that can be applied to the output. Each activation function has its own uses and completely changes the behavior of the output. Instead of defining a Perceptron for each of these functions we can create a more general Perceptron and apply the activation function afterward. Likewise, we'll add the step function after we can compute an output.

Perceptron in Action

Let us use our Perceptron to compute AND, OR gates. Up until now, we haven't worried about what the weights will be and how we can use them to produce the desired output. In order to compute these binary functions, we are going to have to define some values for the weights and the bias. Also, we have to think carefully about how the shape of our inputs, weights, and biases will affect the outputs. We are going to need two different sets of values for creating a Perceptron, one for computing the AND gate and another for the OR gate.

We'll start by representing our inputs as an array of two binary values.

```
inputs = [[0.0,0.0], [0.0,1.0], [1.0,0.0], [1.0,1.0]]
```

Our first Perceptron will compute AND. It shouldn't take long to come up with some values that correctly calculates this logic. Here are the values we found that work nicely for the weights and bias, where $w_1 = w_2 = 1$, and $C = -1$.

```
perceptron = Perceptron([[1.0], [1.0]], -1.0)
```

We can compute the AND gate by looping over the inputs and printing the outputs from the Perceptron.

```
for vec in inputs:
    outputs = perceptron(vec)
    print("I:", vec, "O:", outputs)
```

But when we run this, the following error message appears.

Error Message

```
tensorflow.python.framework.errors_impl.InvalidArgumentError:
In[0] is not a matrix. Instead it has shape [2] [Op:MatMul]
```

These shape errors are an easy mistake to make. The error tells us that In[0] is not a matrix, which is referencing the input we called *vec*, where each *vec* is a one-dimensional array. The error says that these need to have shape [2], which is caused by the function **tf.matmult()**, which we used to multiply the inputs by the weights. This operation requires both Tensors to have a least two dimensions. We can fix this by wrapping each input *vec* inside another array. Then when we run the code, we shouldn't get any errors.

```
for vec in inputs:
    outputs = perceptron([vec])
    print("I:", vec, "O:", outputs)
```

```
Output
I: [0.0, 0.0] O: tf.Tensor([[-1.]], shape=(1, 1), dtype=float32)
I: [0.0, 1.0] O: tf.Tensor([[0.]], shape=(1, 1), dtype=float32)
I: [1.0, 0.0] O: tf.Tensor([[0.]], shape=(1, 1), dtype=float32)
I: [1.0, 1.0] O: tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
```

Yay, we have no errors! Although we can see that without an activation function our values are not giving us a clear signal (i.e. `tf.Tensor([[-1.]])`, and shows why we need one. Most of the provided activation functions in TF are applied element-wise, meaning the function is applied to each element in a Tensor separately. We defined our activation as the step function. Unfortunately, TF does not have this function implemented. And it would take a lot of work for us to add this function as an operation on Tensors. Also, TF will not allow us to iterate over any Tensor in a loop, so we can't apply the step function by iterating over the outputs. Instead we can map the values from a Tensor to a new Tensor using the method **tf.map_fn()**. This function takes two parameters, a function for mapping each value and a Tensor. We already have the output Tensor, so we need to create an activation function for this mapping.

```
Activation Function
def tf_step_fn(a):
    shape_a = a.shape
    a = tf.reshape(a, [-1])
    a = tf.map_fn(step_fn, a)
    return tf.reshape(a, shape_a)

def step_fn(a):
    if a > 0:
        return 1.0
    else:
        return 0.0
```

First, we flatten the Tensor into a 1-D array. Then we apply the mapping and finally reshape the Tensor back to its original shape. This ensures that the mapping with our **step_fn()** is applied element-wise. With this correct step function, we can finally apply our activation function after computing the output for each input.

```
AND Computation
inputs = [[0.0,0.0], [0.0,1.0], [1.0,0.0], [1.0,1.0]]
perceptron = Perceptron([[1.0], [1.0]], -1.5)
for vec in inputs:
    outputs = perceptron([vec])
    outputs = tf_step_fn(outputs)
    print("I:", vec, "O:", outputs)
```

```
AND Output
I: [0.0, 0.0] O: tf.Tensor([[0.]], shape=(1, 1), dtype=float32)
I: [0.0, 1.0] O: tf.Tensor([[0.]], shape=(1, 1), dtype=float32)
I: [1.0, 0.0] O: tf.Tensor([[0.]], shape=(1, 1), dtype=float32)
I: [1.0, 1.0] O: tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
```

Just what we wanted, our Perceptron now correctly computes the AND gate. We'll do the same thing for the OR gate by creating a new instance of **Perceptron** defining its weights and bias.

OR Computation
```
inputs = [[0.0,0.0], [0.0,1.0], [1.0,0.0], [1.0,1.0]]
perceptron = Perceptron([[1.0], [1.0]], -0.5)
for vec in inputs:
    outputs = perceptron([vec])
    outputs = tf_step_fn(outputs)
    print("I:", vec, "O:", outputs)
```

OR Output
```
I: [0.0, 0.0] O: tf.Tensor([[0.]], shape=(1, 1), dtype=float32)
I: [0.0, 1.0] O: tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
I: [1.0, 0.0] O: tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
I: [1.0, 1.0] O: tf.Tensor([[1.]], shape=(1, 1), dtype=float32)
```

With the same activation function and weights, all we did was change the bias achieving a completely different output. It turns out that it is not possible to create a Perceptron to compute XOR. The reason why is because the threshold of the activation function can only separate the inputs into two classes. To compute XOR we need a way to separate the values into at least three classes. To achieve this, we'll have to use multiple Perceptrons.

Linear Separation

To understand why a single Perceptron cannot classify XOR, let us look at how we were using our Perceptrons to linearly separate the inputs for AND, and OR. We can illustrate the classification by representing the input matrix as (x, y) coordinates (see Fig. 5), where the $w_1 = w_2 = 1$.
The Perceptron's classification translates directly to drawing a straight line through this graph, separating the input points. This line is our weighted sum of the inputs plus the bias (from the matrix multiplication) but now as a function of x, y.

$$\text{AND}(x, y) = \begin{cases} 1 & \text{if } x + y - 1.5 > 0 \\ 0 & \text{if } x + y - 1.5 \leq 0 \end{cases}$$

$$\mathrm{OR}(x,y) = \begin{cases} 1 & \text{if } x+y-0.5 > 0 \\ 0 & \text{if } x+y-0.5 \leq 0 \end{cases}$$

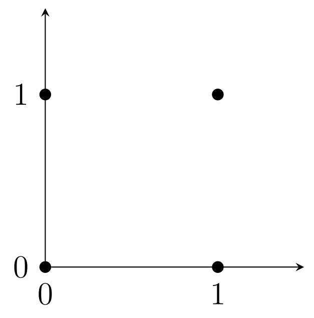
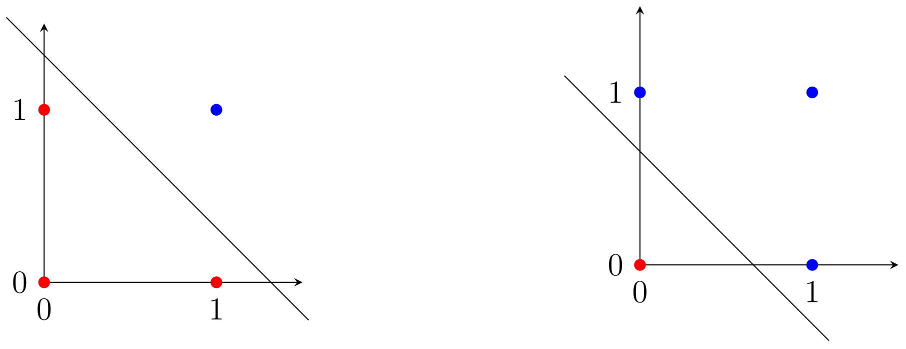


Figure 5: Input Space

Below, these lines are plotted for all x, y. In both cases, we have defined these lines such that they split the inputs into two different classes. The weights and threshold determine the slope and the position of the line on the x-axis, respectively. Additionally, the activation (or threshold) translates to coloring the inputs. When an input is 1, it is colored blue, and when it is 0, it is colored red depending on where the input fits relative to the line. Changing either the weights or the threshold value would result in a new line, hence a different classification.



(a) AND

(b) OR

Figure 6

In either case, we are classifying the inputs based on their features. For the AND case, we want the (0,0) input to be in a different class than the other points. And similarly, for the OR case, we want the (1,1) input to be in a different class.

In the XOR case, when we try to classify the inputs based on their features we run into some trouble. The obvious way to separate the inputs would be with two lines.

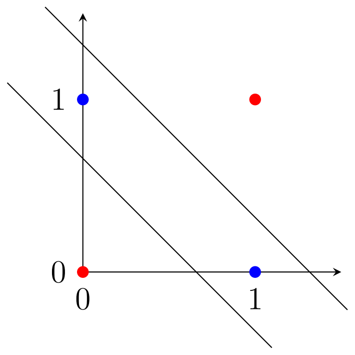


Figure 7: XOR

These two lines separate the inputs into three classes. Since each line represents a Perceptron, we are going to need at least two Perceptrons to solve this problem. However, we only care that the $(0, 1)$ and $(1, 0)$ are put into the same class. We want to treat the other inputs as though they are in the same class even though they are not in the same region. This translates to first putting the lines on the graph and then coloring the inputs with two colors, which means this problem requires two steps to solve. We will show in the section how we can solve this using multiple Perceptrons.

Multi-Layer Perceptron

To build layers of these Perceptrons, we need an easy way to describe the network architecture. So far, we have only looked at networks with two layers: the input and output layers. However, what if we added another layer in between the inputs and the outputs? Conceivably, we could add as many layers as we wish between the input and output layers. These middle layers are usually called the hidden layers. When we add a hidden layer, we simultaneously create an output and an input. Below is a Multi-Layer Perceptron (MLP) diagram with two input layer nodes, one hidden layer with two nodes, and an output layer with one node.

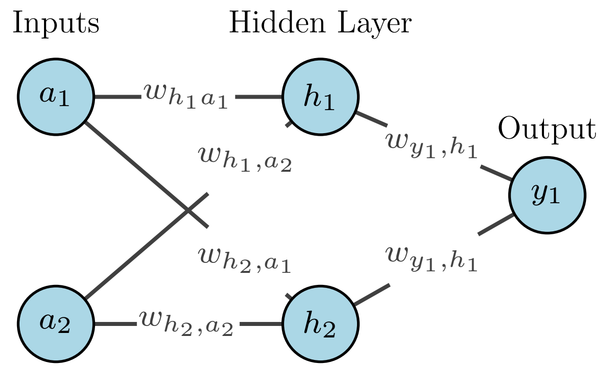


Figure 8

The main difference between this network and the Perceptron is that the hidden layer has two output nodes. Each hidden layer node is a Perceptron, but we can still represent these connections using a single Perceptron with multiple outputs. This way, each input node contributes to every output node in the next layer. We reference these weights by their associated layer connections. So the weight between a_1 and h_1 is denoted as w_{h_1,a_1}.

To compute an output, we first need to feed the inputs to the network and calculate the hidden layer nodes. Just like before, we do this with matrix multiplication, adding the bias, and applying the activation function δ, so that hidden nodes values are between zero and one.

$$\begin{bmatrix} w_{h_1,a_1} & w_{h_1,a_2} \\ w_{h_2,a_1} & w_{h_2,a_2} \end{bmatrix} * \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \rightarrow \delta \left(\begin{bmatrix} (w_{h_1,a_1} * a_1) + (w_{h_2,a_2} * a_2) + b_1 \\ (w_{h_2,a_1} * a_1) + (w_{h_2,a_2} * a_2) + b_2 \end{bmatrix} \right) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

Then this hidden layer output becomes the input for computing the final output.

$$\begin{bmatrix} w_{y_1,h_1} & w_{y_1,h_2} \end{bmatrix} * \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \begin{bmatrix} b_3 \end{bmatrix} \rightarrow \delta \left(\begin{bmatrix} (w_{y_1,h_1} * h_1) + (w_{y_1,h_2} * h_2) + b_3 \end{bmatrix} \right) = \begin{bmatrix} y_1 \end{bmatrix}$$

Note, the output nodes can only be computed after the hidden layer nodes are calculated. We can think of these operations as a sequence of steps which combined together to compute the output. For any MLP, the output is always computed layer by layer starting at the input layer and finishing at the output layer.

Multi-layer Perceptron in TensorFlow

Since we defined our Perceptron class in TF to be generic, we easily translate this MLP and its matrix operations into Tensor operations.

MLP class

```
class MLP(tf.Module):
    def __init__(self, w1, b1, w2, b2, name=None):
        super().__init__(name=name)
        self.input_hidden = Perceptron(w1, b1)
        self.hidden_output = Perceptron(w2, b2)
    def __call__(self, x):
        h1 = self.input_hidden(x)
        h1 = tf_step_fn(h1)
        output = self.hidden_output(h1)
        output = tf_step_fn(output)
        return output
```

This code should feel very similar to the Perceptron. The structure of the class is the same. But now, we are creating layers of Perceptrons instead of Tensors. Once again, our mutable data lives inside the constructor consisting of two Perceptrons, each with its own sets of weights and biases. We use the MLP function to define the layer-by-layer computation of the output so that the inputs are processed how we defined it using our MLP diagram and its matrix operations.

First, we send the data through the input-to-hidden Perceptron, which computes each node's output in the hidden layer. Then we apply the activation function and feed that through the hidden-to-output Perceptron, where finally we apply the same activation function and return the output.

Simple right? Well, this will only work if our defined weights and biases compute the correct shapes at each step in the process. If we not mindful of this, we might get some scary shape errors. Our class assumes that Tensor operations will work out and cause no errors. However, we should be able to easily avoid this since we have already defined the matrices that represent each set of weights and biases. Before we do an example, let us define each parameter's shape so that our MLP does not yell at us.

Using the matrix representation of our MLP, we know that the inputs to the MLP will have a shape of (1,2). For the input to hidden Perceptron, the weights must have the shape (2,2), and the bias must have a shape (1,2). Then we know the output shape of this Perceptron must be $(1, 2) * (2, 2) = (1, 2)$. Luckily the activation function does not change the shape of our outputs, so nothing needs to be done. We know that the inputs will have the shape (1,2). The weights must have the shape (2,1), and the biases must have a shape (1,1). Therefore, the final output will have the shape $(1, 2) * (2, 1) = (1, 1)$.

Multi-Layer Perceptron Example

With this in mind, we can use our MLP to compute the XOR function. Again, we will define our inputs as a list of binary values. Then we can compute their logic tables by looping over each input and feeding it to the MLP, producing an output. However, this time we have a lot more values to define so that the MLP correctly classifies each input. The only way to do this at the moment is by writing out a Perceptron and trying out different values for each weight and bias. This will be the last time we have to do this, but it is a good exercise for understanding how each part in the MLP contributes to the output. Here are the values we found that correctly compute XOR.

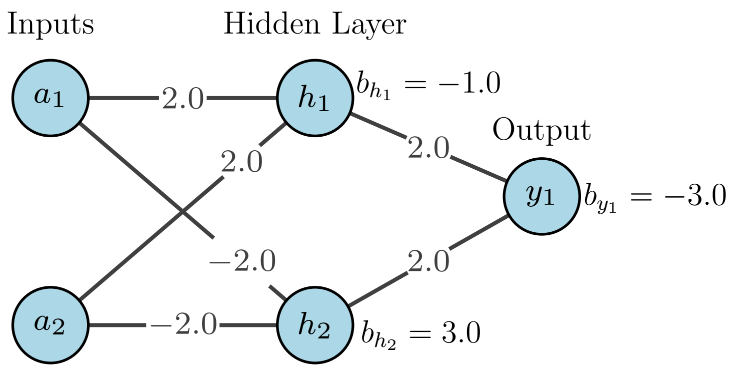


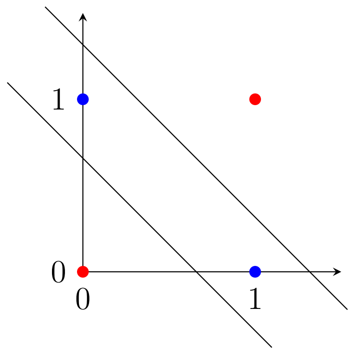
Figure 9

Let us walk through how these values in the MLP correctly compute XOR. Before, when we looked at our inputs as coordinates, we said that XOR would need at least two Perceptrons to solve the problem. This was represented by the two lines that separated the inputs so that $(0, 1)$ and $(1, 0)$ were in the same class. Those two Perceptrons are the hidden layer nodes. Each hidden layer node plays a different role in separating the inputs into the three classes. We can illustrate this by looking at each hidden layer output for every input.

(a_1, a_2)	h_1	h_2	y_1
$(0, 0)$	0	1	0
$(0, 1)$	1	1	1
$(1, 0)$	1	1	1
$(1, 1)$	1	0	0

Notice that the inputs $(0, 1)$ and $(1, 0)$ are in the same class since they both output $[1, 1]$ for the hidden layer. Whereas the inputs $(1, 1)$ and $(0, 0)$ output different values at the hidden layer, meaning they are in different classes. We can see this by looking back at our (x, y) coordinate representation of the inputs and the two lines that separate them into three different classes.

To understand how each hidden layer node draws these lines, we can look at their behavior separately. When we examine the outputs of the h_1 node, we can see that



XOR

it is acting as an *OR* gate, while the outputs of the h_2 node are acting as a *NAND* gate. This is not a coincidence, and if we were to compute the outputs manually, we would get these exact values shown in the table above.

Finally, to classify our inputs into only two classes and not three, we can use the last layer as an *AND* gate. The purpose of this last layer is to color the graph's points with only two colors. Since our inputs to this layer are either $(0,1)$, $(1,0)$, or $(1,1)$ an *AND* Perceptron will correctly output a one in the case of $(1,1)$ and a zero in the other case. Then we will end up with a network that correctly computes XOR. Now let us implement the MLP in TF and solve for the XOR logic table.

XOR Computation

```
inputs = [[0.0,0.0], [0.0,1.0], [1.0,0.0], [1.0,1.0]]

w1 = [[2.0, -2.0],[2.0,-2.0]]
b1 = [[-1.0, 3.0]]
w2 = [[2.0],[2.0]]
b2 = [[-3.0]]

mlp = MLP(w1, b1, w2 , b2)

outputs = mlp(inputs)
print("XOR: ", outputs)
```

```
XOR Output
XOR:  tf.Tensor([[0.] [1.] [1.] [0.]], shape=(4, 1), dtype=float32)
```

We removed the loop for computing the output because it trims down the code into one line and leverages Tensor operations. We said that the outputs should have a shape (1,1), but this was for a single output. Since our MLP matrix operations only relied on the rows of the inputs being equal to the weights' columns, we are free to add as many columns to the inputs as we wish. Since our inputs now have a shape of (4,2), we can then expect the output of the MLP to have a shape (4,1). This essentially allows us to compute all of the outputs at once (under the hood, we know they are still being computed one by one through matrix multiplication). Our outputs are correct; we showed why the weights and biases work, and we even predicted what the output shapes should be.

The Two-layer Multi-Layer Perceptron

That was a lot of hard enough work. However, it will be the last time we ever have to think about the actual values inside an MLP. So far, we have only solved binary classification problems, which for both the MLP and the Perceptron required two inputs and a single output. If we want to classify handwritten digits, we will need a larger network to fit the input (28x28 pixel images) and output space (10 digits). Let's generalize the two-layer Multi-Layer Perceptron for n inputs, m hidden nodes and k outputs, as shown in Figure 10.

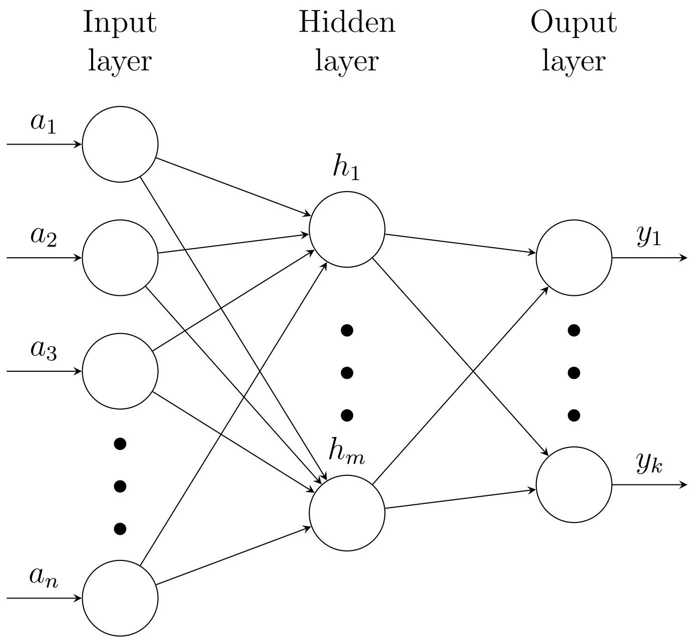


Figure 10: Graphic altered from Mark Wibrow's code [2]

Then, we can define the MLPs weights and biases with respect to its layer connections. First we will define the matrices for the input-to-hidden layer:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}, \mathbf{w}_1 = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

Next, we will defined the matrices for the hidden-to-output layer:

$$\mathbf{w}_2 = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1} & w_{k2} & \cdots & w_{km} \end{bmatrix}, \mathbf{b}_2 = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}.$$

To compute the output of the MLP, we can define its mapping as the function $M\colon A \to Y$, where A is the domain of real input matrices and Y is the range of real output matrices. We can break down this function layer by layer and then combine them using function composition.

Let the input-to-hidden layer be,

$$g(\mathbf{a}) = \alpha(\mathbf{w}_1 * \mathbf{a} + \mathbf{b}_1)$$

where α is an arbitrary activation function. Then let the hidden-to-output layer be,

$$h(g(\mathbf{a})) = \alpha(\mathbf{w}_2 * g(\mathbf{a}) + \mathbf{b}_2) = \mathbf{y}$$

Then $M(\mathbf{a}) = h(g(\mathbf{a})) = \mathbf{y}$. Now we have a mathematical way of representing our two-layer network in terms of matrix operations for an arbitrary network.

Now that we have generalized our MLP, it is clear that the MLP is a mapping between the inputs and the outputs. This mapping constructs the relationship between the inputs and their classification as a mathematical equation. In our case, we are mapping the handwritten digit images to their appropriate labels. If we want to classify handwritten digits, we will need to separate our inputs into ten different digits. This means we need an output node for every possible digit. Since the inputs will be the image pixels, we will then have a matrix of weights for each pixel connected to some number of hidden layer nodes. Then each hidden layer node will be connected to the ten output nodes. With so many connections, it would be impossible to set every value for the network's weights and biases. To get around this, we are going to use a supervised learning method known as backpropagation. This learning algorithm will optimize our mapping M by incrementally updating a random set of weights and biases proportionally to the MLP's output. In the end, we will have an MLP whose function M will very accurately map the images to their respective labels.

This approach of changing the weights and biases incrementally relative to the output is the core of how the MLP learns. At first, our network's weights and biases will be set randomly, meaning our initial outputs will be pretty bad. The backpropagation algorithm uses the image label as a metric for determining the new weight and bias. To understand how this will happen, we need to make one final change to our implementation of the two-layer MLP.

Sigmoid Activation Function

Since we will not be manually changing the weights and biases, we have to tell the MLP how to do this. Let us define the set of weights and biases as W and B

respectively to focus on their relationship to the output. When updating the weights W and biases B, we want the small changes to their values to result in a small change to the actual output $\mathbf{y}$. The change in these values will be denoted as ΔW, ΔB, and $\Delta \mathbf{y}$.

For example, imagine at one step in the training process giving the MLP a handwritten digit of a nine, and let us say the prediction comes out as an eight. So whichever weights and biases contributed the most to this incorrect output should have its values changed by different ΔW's and ΔB's. These values are proportional to how far away the actual output is from the desired output. We want these changes to the weights and biases to be small so that the learning process can be more sensitive to the inputs' small differences.

(a)

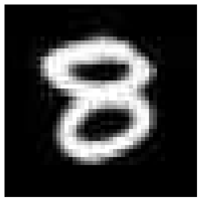
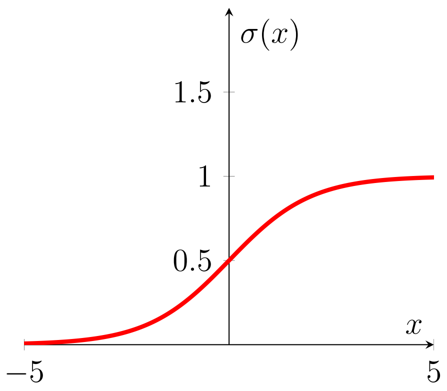
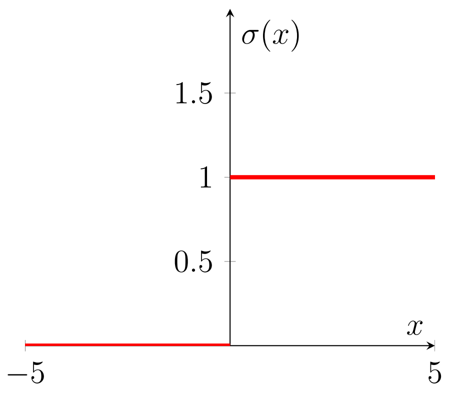

(b)

Figure 11: MNIST Images

However, we will not be able to achieve this behavior using our step activation function. A small change to weights and biases might change the output from one to a zero. Using our example, this would have the effect of making the MLP better at predicting a 9. Now imagine we gave the MLP a handwritten digit of an eight, and it changed that output back to a one from a zero. This behavior makes the MLP more likely to predict outputs for the input that it has seen the most. The step activation function does not allow the MLP to represent all the possible inputs and predictions accurately. We can solve this issue by choosing a different activation function that allows for more precise tuning of the weight and biases. We can achieve this behavior with the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ where x is an arbitrary input.



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



$$\delta(y) = \begin{cases} 1 & \text{if } W * x + B > 0 \\ 0 & \text{if } W * x + B \leq 0 \end{cases}$$

Figure 12

The main issue with the step function is that any input x close to zero is the same as another input far away from zero. In comparing the two functions, the sigmoid smooths out this behavior for the outputs between zero and one. Due to its asymptotic behavior, inputs farther away from zero get closer and closer to the max range of outputs, one and zero. With the sigmoid, the inputs closer to zero behave differently than the inputs farther away. This means that gradual changes to the weights and biases (the inputs) will correspond to gradual changes in the outputs.

There are other functions with this asymptotic behavior, such as a hyperbolic tangent function. However, the sigmoid function's main attraction is that it is differentiable, and the derivative is easy to calculate. On the other hand, the step function is not differentiable because it is discontinuous. The nature of the step function does not understand the relationship between the inputs and the outputs. Since the sigmoid is differentiable then we can define this relation using its derivative, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, which we will verify in the Backpropagation section.

This derivative will be used in the backpropagation algorithm for finding the ΔW's and ΔB's depending on the output $\mathbf{y}$. In the end, the sigmoid function will allow the algorithm's small changes to the weights and biases to correspond to small changes in the outputs.

Naturally, TF implements this exact function (and its derivative), and just like our step function, it is applied element-wise. We can update our MLP class one last time by replacing its activation function with the sigmoid TF operation. This will be the class that we use to create an MLP for classifying handwritten digits.

Final MLP class

```python
class MLP(tf.Module):
    def __init__(self, w1, b1, w2, b2, name=None):
        super().__init__(name=name)
        self.input_hidden = Perceptron(w1, b1)
        self.hidden_output = Perceptron(w2, b2)
    def __call__(self, x):
        h1 = self.input_hidden(x)
        h1 = tf.math.sigmoid(h1)
        output = self.hidden_output(h1)
        output = tf.math.sigmoid(output)
        return output
```

XOR with Sigmoid

For clarity, let's run our XOR example with this final version of the MLP to see how the sigmoid changes our outputs.

XOR Sigmoid

```python
inputs = [[0.0,0.0], [0.0,1.0], [1.0,0.0], [1.0,1.0]]
w1 = [[2.0, -2.0],[2.0,-2.0]]
b1 = [[-1.0, 3.0]]
w2 = [[2.0],[2.0]]
b2 = [[-3.0]]
mlp = MLP(w1, b1, w2 , b2)
outputs = mlp(inputs)
print("XOR: ", outputs)
```

Output

```
XOR:  tf.Tensor(
[[0.3642491 ]
```

```
 [0.48106766]
 [0.48106766]
 [0.3642491 ]], shape=(4, 1), dtype=float32)
```

Instead of the outputs being only zero or one, we now have a smoother range of values. Even though the outputs are not technically binary, we are still getting a binary classification. We can make this clearer by scaling our weights and biases by a value of 100 so the outputs are closer to the asymptotes of the sigmoid function.

```
New Weights and Biases
w1 = [[200.0, -200.0],[200.0,-200.0]]
b1 = [[-100.0, 300.0]]
w2 = [[200.0],[200.0]]
b2 = [[-300.0]]
```

```
Output
XOR:  tf.Tensor(
[[0.]
 [1.]
 [1.]
 [0.]], shape=(4, 1), dtype=float32)
```

Since scaling is a linear operation, we get the same behavior with different output values. Scaling our weights and biases changed the difference between 0.3 and 0.4 by 100 percent. Then by the asymptotic behavior of the sigmoid, our values became 0 and 1. This illustrates how much control we have over the range of outputs by changing the weights and biases. The asymptotes of the sigmoid allow for the network to have a sense of confidence. If the output is closer to zero, the network is less sure of its classification. If the output is larger than, say, 100, then the network is very confident of its classification. This behavior ensures that the backpropagation algorithm will work. The metric that we will use for determining the accuracy of a network will be dependent on this behavior.

Recognizing Handwritten Digits

Now that we have all the pieces to understanding the MLP model, we can start solving the more challenging problems. In the following sections, we will walk through building an MLP to recognize handwritten digits. To do this, we will need a data set, a model (the MLP), and a training algorithm. Let us start by loading in our data set and building the model.

Data Set and MLP Model

First, we need to load in the MNIST data set. As mentioned before, this is a collection of labeled images of handwritten digits. The MNIST data set contains 60,000 labeled images for training and 10,000 labeled images for testing. There is nothing special about the number of images in each collection other than making it easier to benchmark machine algorithms using this convention.

TensorFlow already contains this data set in the API. We can load it using another API built on top of TensorFlow called Keras. Also, to help us see what the images look like, we need to import a rendering API. Now, we will start by importing these APIs at the top of our file and loading the MNIST dataset (see Appx. pg. 80 for details on installation).

SET UP

```
import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt

(train, test), (lbtrain, lbtest) = keras.datasets.mnist.load_data()
```

The images are named: **test** and **train**, while their respective labels are named: **lbtrain**, and **lbtest**. To see the Tensor objects, we have been printing them out by directly calling the Tensor inside the print statement. Usually, we would see the value of the Tensor, its shape, and type. However, if we did that with these Tensors, our command line would spit out some representation of all the thousands of images or labels. For now, we only need to know the shape and type. We can do this using the built-in properties (known as “attributes”) for Tensors.

Training Set Properties
```
print("TRAIN SHAPE: ", train.shape)
print("TRAIN TYPE: ", train.dtype)
print("LBTRAIN: SHAPE", lbtrain.shape)
print("LBTRAIN TYPE: ", lbtrain.dtype)
```

Output
```
TRAIN SHAPE:  (60000, 28, 28)
TRAIN TYPE:  uint8
LBTRAIN: SHAPE (60000,)
LBTRAIN TYPE:  uint8
```

We can glean from our training data’s output shape that it is a list of 60,000 Tensors, where each Tensor has the shape (28,28), representing the dimension of the image by its width and height in pixels. The **dtype** of the Tensor representing the image is a **uint8** which is an integer value representing the greyscale intensity of the pixel as a value between 0-255. On the other hand, the labels are a list of 60,000 Tensors, which are integer labels for each image as an **uint8**. For the testing images, we would see identical Tensor shapes and types but only with 10,000 entries.

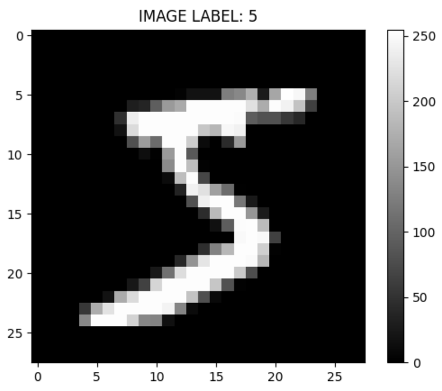
Now that we have our data set loaded let us take a closer look at one of these training images. We can pick out a single image by indexing the first image and its label from the training set. Then, using **pyplot**, we can quickly render the greyscale image and its label. However, we have to convert the label from a **uint8** to a string so it can be appropriately displayed.

RENDER IMAGE

```
img = train[0]
img_label = lbtrain[0]

plt.figure()
plt.imshow(img, cmap='gray')
plt.colorbar()
plt.title("IMAGE LABEL: " + str(img_label))
plt.grid(False)
plt.show()
```

From the render on the right, we can see that the image is a grey-scale 28x28 pixel image, where each pixel value ranges from 0 (black) to 255 (white). Also, after the conversion, we see that this is indeed the correct label for this image. This is great! We have our dataset, and we know how to store images as Tensors. However, we are going to need to do some pre-processing of the data. To see why let us build our MLP by defining its network architecture.



MLP Model

In the API, TensorFlow refers to a network's architecture and data flow as a model. This is a more generic term that encompasses different model designs. In our case, the model is the MLP that will recognize handwritten digits. The model becomes a neural network once it is trained and tested. The TensorFlow model pipeline emphasizes creating a fully integrated model before training and evaluating. We have to ensure that the number of layers and nodes in our MLP model represents how our data is processed. So far, we have only looked at an MLP with three layers. In the case of XOR, we designed our model to take the two possible inputs and produce a single binary output. If we want our model to recognize handwritten digits, we must first be able to input the image and output a classification. Below are the specifications of the three-layer model that will use to solve this problem.

- **Input Layer:** 784 input nodes
- **Hidden Layer:** 15 hidden nodes
- **Output Layer:** 10 output nodes.

Let us start with the input nodes. Since our images are 28x28 pixels, there are a total of 784 pixels, one for each input.

Then, we create ten nodes for the output nodes, where each node represents the ten possible labels. We do this because we want the model to have the ability to predict any of the ten labels. We did not do this for the XOR example because we knew how the weights and biases would affect the output. Also, our classification was binary, and it was sufficient only to have one output. But now that we have ten categories, we need them all present within an output since it is not practical to initially set the weights and biases.

Each of these outputs will be a value between 0 and 1 because of the sigmoid activation function. We can map these values to a label by choosing the index for each value in the ten outputs. This means our output Tensor will represent in order by index each digit from 0,1,...,9. Initially, all of the weights and biases will be set at random for our model, giving us a good starting point for optimizing the weights and biases. In the beginning, every classification will have an output where each value could be any number between one and zero, having the effect of our prediction being total nonsense. However, we can still take these outputs and produce a single prediction by taking the largest of these values. Whichever output value was the largest for that index 0,1,...,9 becomes the single prediction of the model.

Finally, it is time to answer the question behind the hidden layer nodes. There is no exact science to know how many hidden layers to use. In our case, one hidden layer to enough to solve this problem without being too computationally demanding.

However, we can be more clever in choosing the number of hidden layer nodes we use. Thinking back to the XOR MLP, we showed how having 2 hidden nodes was essential in separating the inputs into three classes. If we only had one hidden layer node, it would not be possible for that model to achieve the classification. So, we used two nodes to extract the inputs' different features since they were not linearly separable.

For the image recognition MLP, if we do not have enough hidden layer nodes, we will run into a similar problem where our model will not classify all possible inputs. It is a bit harder to determine these "features" of our images, but we should have a least 10 hidden nodes representing the 10 possible digits as a minimum. Any fewer would overgeneralize the main classification problem and limit the possible outputs the model can produce.

On the other hand, if we use too many hidden layer nodes, our MLP is more susceptible to over-training, which happens when the number of "features" is larger than the number of inputs. With too many hidden nodes, the MLP loses the ability to generalize the inputs for a given category. There is an extreme case where having too many hidden nodes results in memorizing every training input.

So we want to strike a balance between the two extremes, allowing the MLP to learn the distinct features of each category and generalize the minute differences between each input for a category. In the end, finding this balance is something to play around with to achieve better results; but for now, we will use 15 hidden layer nodes as a starting point.

Now that we have defined our model architecture, we have to ensure that the model reflects the operations that contribute to the output. Let us pre-process our data to input an image and verify that the model is working.

Pre-Processing MNIST

Let us start by altering our images so that they can be the inputs to our model. First, we need to flatten every image from its 28×28 shape Tensor into a flat 1×784 shape Tensor, which we can do with the reshape operation. We saw that the image has pixel values between 0 and 255 represented by a **uint8**, which in our model will not work. This is because the MLP expects output values between zero and one represented as a **float32**. We can fix this problem by re-scaling the inputs and divide each pixel value by 255 so that our values range between zero and one. Since these values are fractional, we need a **float32** to represent it. So we must convert the image pixels type from a **uint8** to a **float32** after the division. Also, since our weights and biases will be **float32**'s, we do this to ensure that all the operations inside the MLP are defined between the same type. With these changes, our images will be good to go.

Next, we need to translate our single label into a form that is similar to our outputs. We cannot define a metric for how bad our outputs are if the labels are not directly comparable. As an example, if we had the label 8, we would then represent this as a Tensor in the form, $[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]$. We can do this every easily by utilizing the Keras API, which has a function that translates numerical data into categorical data represented in the form we just described. When we do this, we have to declare the type. Since our outputs will be **float32**'s, when we do this translation, each value must be a **float32**'s so that we can perform operations on it. With this change, our labels will be ready for the training process.

PRE-PROCESS DATA

```
train = train.reshape(60000, 784).astype("float32") / 255
test = test.reshape(10000, 784).astype("float32") / 255

lbtrain = tf.keras.utils.to_categorical(lbtrain, num_classes=10,
                                                dtype="float32")
lbtest = tf.keras.utils.to_categorical(lbtest, num_classes=10,
                                                dtype="float32")
```

Building the Model

Since our MLP class allows us to express any model architecture, we need to create the Tensor for the weights and biases according to our model for recognizing handwritten digits. Initially, we choose to set these values randomly. TF has several random number generators that we can utilize to populate a Tensor of any dimension we specify. We will use TF's uniform distribution implementation bounded by $(-1, 1)$ to ensure that our values start small and evenly distributed. This function takes four parameters: shape, minimum value, maximum value, and type.

PRE-PROCESS DATA
```
inputs_hidden_w = tf.random.uniform([784,15], -1,1, dtype="float32")
inputs_hidden_b = tf.random.uniform([15], -1,1, dtype="float32")
hidden_outputs_w = tf.random.uniform([15,10], -1,1, dtype="float32")
hidden_outputs_b = tf.random.uniform([10], -1,1, dtype="float32")
```

Now we can pass these values into our MLP and feed in a single input image to see what happens.

MLP EXAMPLE
```
mlp = MLP(inputs_hidden_w, inputs_hidden_b,
            hidden_outputs_w,hidden_outputs_b)

img = train[0]

out = mlp([img])
print("OUTPUT: ", out)
```

MLP OUTPUT
```
 OUTPUT:  tf.Tensor(
[[0.9671001  0.995833   0.9893452  0.05819231 0.05141053 0.61459994
  0.9345007  0.9493144  0.49852577 0.242275  ]],
  shape=(1, 10), dtype=float32)
```

Looks good! These outputs will differ every time we create a new MLP since they are randomized. But looking at these outputs, we can see that the largest value is 0.995833, which corresponds to the digit 1. Looking at the other digits, a few values from this output are also very close to one. We can interpret this by saying that the MLP is not confident in knowing which digit to predict. In the next section, we will use this output as motivation for implementing the backpropagation algorithm.

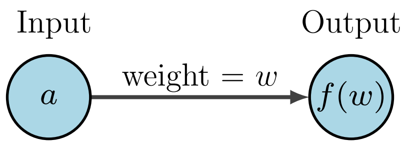
Backpropagation

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.

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

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

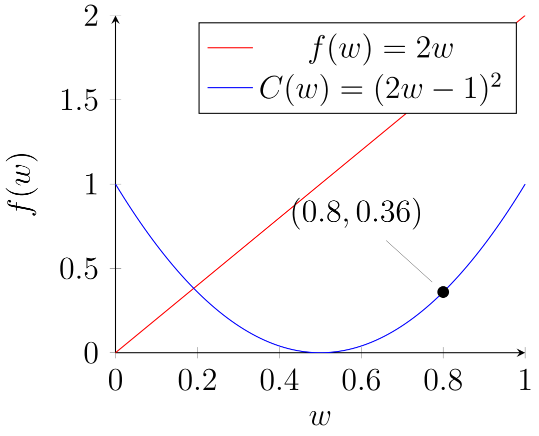
For the fixed input $a = 2$, this network outputs 1.6, but we want $f(w) = 1$. The goal is to figure out what value for w will output 1. We will start by considering the squared distance between the actual output $f(w)$ and the desired output $\hat{y}$:

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

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

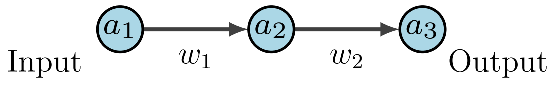
```python
def update_weights(updates, lr=1.0):
    for i in range(updates):
        dw = cost_partial()
        dw2 = tf_partial()
        assert(dw == dw2)
        w.assign_sub(lr * dw)
```

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.

The Multi-Layered Network

We can take our simple network and add multiple layers. For this example, we'll call our fixed input a_1, create a single hidden node a_2, and an output node a_3.



As we add more notation, we cannot forget that the output is a function of all the previous weights and nodes. Thus,

$$a_3 = a_2 * w_2 = (a_1 * w_1) * w_2$$

We will not be writing the output in this form. However, it will be helpful to think about these functions when we need to compute the partial derivatives to update the weights. Except for the input a_1, we can write all the a's as a function of the previous node and weight.

$$a_3 = a_2 * w_2$$

$$a_2 = a_1 * w_1$$

Now that we have broken down the forward propagation, we can optimize this network using gradient descent. Again, we want to minimize the cost function for our fixed input a_1 to produce the desired output $\hat{y}$. The cost function looks the same as before, but now, more weights are contributing to the output of the network a_3.

$$C(w_1, w_2) = (a_3 - \hat{y})^2$$

Since the output function a_3 depends on all the weights, we need to compute the partial derivative of the cost function with respect to each weight, where r is the learning rate. Thus the weight updates are as follows,

$$w_2^{'} = w_2 - r\frac{\partial C}{\partial w_2} = w_2 - r\frac{\partial a_3}{\partial w_2}\frac{\partial C}{\partial a_3} = w_2 - r(a_2)(2(a_3 - \hat{y}))$$

$$w_1^{'} = w_1 - r\frac{\partial C}{\partial w_1} = w_1 - r\frac{\partial a_2}{\partial w_1}\frac{\partial a_3}{\partial a_2}\frac{\partial C}{\partial a_3} = w_1 - r(a_1)(w_2)(2(a_3 - \hat{y}))$$

The backpropagation algorithm always starts with the last layer and propagates the partial derivatives backward through each weight. At each step, we compute the partial derivative of the cost function with respect to the previous weight. To illustrate this, process let us expand on the example in the previous section by adding another weight. Consider the following the network where $\hat{y}$ is the desired output,

$$a_1 = 2, w_1 = 0.8, w_2 = 3.0, \hat{y} = 1$$

$$a_3 = 2(0.8)(3.0) = 4.8$$

To illustrate the backpropagation for this network, we can refactor our existing code from the previous section for updating multiple weights. We will start by defining the network and the cost function.

MULTILAYERED NETWORK

```
a1 = tf.Variable(2.0)
w1 = tf.Variable(0.8, trainable=True)
w2 = tf.Variable(3.0, trainable=True)
d = tf.Variable(1.0)

def a3(a1, w1, w2):
    return a2(a1,w1) * w2

def a2(a1, w):
    return a1 * w

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

To verify that TF is calculating the same partial derivatives, we will hard-code these partial derivatives with respect to each weight which we derived.

PARTIAL DERIVATIVE FUNCTIONS

```
def cost_partial_w2():
    return a2(a1,w1)*2*(a3(a1,w1,w2) - d)
```

```
def cost_partial_w1():
    return a1*w2*2*(a3(a1,w1,w2) - d)
```

Then, using *tf.GradientTape()* we can calculate the same partial derivatives. In this example, we need to find $\frac{\partial C}{\partial w_1}$, and $\frac{\partial C}{\partial w_2}$. Since the weights are set as trainable, we can list each weight we want to find the partial derivative of the cost function C with respect to, in the second argument of *tape.gradient()*. Once these are calculated by the *tape*, the function will return a corresponding list of the partial derivatives. Any collection of partial derivatives is known as a gradient.

TF PARTIALS

```
def tf_partial():
    with tf.GradientTape() as tape:
        c = cost(a3(a1, w1, w2), d)
        [dw1, dw2] = tape.gradient(c, [w1, w2])
    return [dw1, dw2]
```

Now that we have both implementations for calculating the partial derivatives, let us refactor our weight update function.

WEIGHT UPDATES

```
def update_weights(updates, lr):
    for i in range(updates):
        print(i)
        dw2 = cost_partial_w2()
        dw1 = cost_partial_w1()
        [tf_dw1, tf_dw2] = tf_partial()
        assert(tf_dw1 == dw1)
        assert(tf_dw2 == dw2)
        w1.assign_sub(lr * dw1)
        w2.assign_sub(lr * dw2)
```

```
update_weights(25, lr=0.01)
print("w1': ", w1, "w2: ", w2, " a3: ", a3(a1,w1, w2))
```

Again, the goal of the updates is to optimize each weight so that the network outputs the desired value $\hat{y}$. In this example, we are starting with a network that outputs $a_3 = 4.8$, and we want it to output $\hat{y} = 1$. Additional, we need a value for our learning rate, so let us set $r = 0.01$ such that our changes to the weights are even more gradual than before.

```
NETWORK AFTER WEIGHT UPDATES
w1:  <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.17478575>
w2:  <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=2.8606453>
a3:  tf.Tensor(1.0, shape=(), dtype=float32)
```

We can see after 25 weight updates that our output is one! It is important to note that we did not get any assertion errors from the updates, meaning that the partial derivatives we derived are the same as the partials given by TF. We hard-coded these partials to illustrate that they are straightforward calculations. We could do this for larger, fully connected networks. However, they would only be limited to find the partial derivatives for a given function. The power of TensorFlow's implementation allows us to easily change the cost or output function without having to hand-calculate the partial derivatives all over again. We can really think of the function *tape.gradient(C, [w])* as the partial derivative of $\frac{\partial C}{\partial w}$, where w is any well-defined Tensor. Therefore, we will use TensorFlow's gradient calculator when implementing our backpropagation algorithm. In the following sections, we will derive the gradients necessary in our MLP for recognizing handwritten digits so that we can eventually update the 11,935 weights and biases.

The Simple Example Revisited

In the previous examples, we excluded the bias and activation function on purpose. But if we want to apply this algorithm to our MLP, we must include them. The only difference these two operations make in the calculations is the complexity of the partial derivatives. This section will redo our simple network with just one layer considering the new variables to show how to compute these networks.

Let b be the bias, $\sigma(z)$ be the sigmoid function, r be the learning rate, and $\hat{y}$ be the desired output. Consider the new network, where the output is defined as follows,

$$z = a * w + b$$
$$y = \sigma(z)$$

We can substitute the output into the cost function, where,

$$C(w) = (y - \hat{y})^2 = (\sigma(a * w + b) - \hat{y})^2$$

To minimize the cost function, we must find $\frac{\partial C}{\partial w}$, which we can write in the form,

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

In the section on the sigmoid activation function (insert link here), we said the derivative of $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. We will now show this derivation.

We know that, $\sigma(z) = \frac{1}{1+e^{-z}}$. We will start by using u-substitution to split up the derivative.

$$u = 1 + e^{-z}$$
$$\sigma'(z) = \left(\frac{d}{du}\frac{1}{u}\right)\frac{du}{dz}$$

Then, we can calculate each part of the derivative separately and substitute back into the equation.

$$\frac{d}{du}\frac{1}{u} = -\frac{1}{u^2}$$
$$\frac{d}{dz}[u] = \frac{d}{dz}[1 + e^{-z}]$$
$$\frac{du}{dz} = -e^{-z}$$
$$\sigma'(z) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

To get this in form we want, we can rewrite this derivative by expanding the expression and cleverly adding zero.

$$\begin{aligned}
\sigma'(z) &= \frac{e^{-z}}{(1+e^{-z})^2} = (\frac{1}{1+e^{-z}})(\frac{e^{-z}}{1+e^{-z}}) = \sigma(z)(\frac{e^{-z}}{1+e^{-z}}) \\
&= \sigma(z)(\frac{e^{-z}+1-1}{1+e^{-z}}) = \sigma(z)(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}) \\
&= \sigma(z)(1 - \frac{1}{1+e^{-z}}) \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}$$

Now that we have shown, $\sigma'(z) = \sigma(z)(1-\sigma(z))$, we can substitute it into the partial derivative of the cost function.

$$\frac{\partial C}{\partial w} = a * \sigma'(z) * 2(y-\hat{y}) = a * \sigma(z)(1-\sigma(z)) * 2(y-\hat{y})$$

Once again we would use this equation to update the weights using the learning rate r, where,

$$w' = w - r * \frac{\partial C}{\partial w} = w - r * a * \sigma(z)(1-\sigma(z)) * 2(y-\hat{y})$$

As a result, we now know how to calculate the partial derivative using the activation function and bias. A nice thing about the derivative of the sigmoid $\sigma'(z) = \sigma(z)(1-\sigma(z))$ is that $\sigma(z)$ is already calculated as the output of the network. This speeds up calculating the partial derivative with respect to the weight since we do not have to re-evaluate each weight update's output. When we compute the partial derivative with respect to the weight, the bias will never be used in the calculation (except for the output $\sigma(z)$) since it is a constant. However, the weights are not the only thing that is updated in the backpropagation algorithm, and we also have to update the biases. We have already done all of the hard work so that we can update the biases through the same process.

For the same network, let us calculate the bias update. We will now take the derivative of the cost function with respect to the bias.

$$\frac{\partial C}{\partial b} = \frac{\partial z}{\partial b}\frac{\partial y}{\partial z}\frac{\partial C}{\partial y}$$

$$\frac{\partial C}{\partial b} = \sigma(z)(1 - \sigma(z)) * 2(y - \hat{y})$$

Since the bias is constant, then its derivative is always one. Therefore, any calculation on the weights can be used to find the partial derivative of the bias. Hence, an update to the bias will be similar to that of the weight.

$$b' = b - r * \frac{\partial C}{\partial b}$$

In conclusion, adding the weights and biases did not change the overall process. It just changed how the partial derivatives were calculated. Now that we have all the pieces of the partial derivatives for a single node in the MLP, we can generalize these equations for a fully connected network.

The Fully Connected Network

We have seen the backpropagation algorithm and gradient descent in action with simpler networks. In this section, we will generalize these examples. Before we jump into the fully connected network, we will split apart the two primary operations for one layer in an MLP. These two operations are the row and column calculations performed in the forward propagation (*see matrix operations, pg. 24*). We will start with the row operations for a single-layer network and calculate its partial derivatives.

Consider a single layer network with one input node a_1, and n output nodes, where $z_n = a_1 * w_n + b_n$.

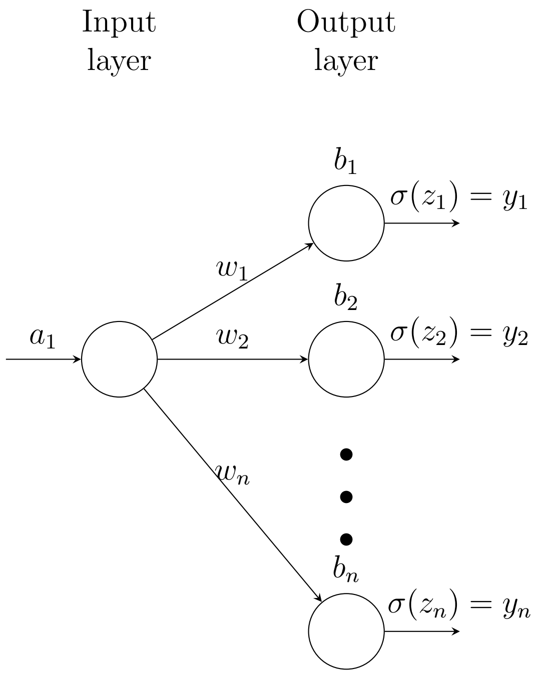


Figure 13: Single Input Neuron [2]

We can start the backpropagation algorithm, by writing our weights, biases and outputs as vectors.

$$\mathbf{w} =< w_1, w_2, \ldots, w_n >, \mathbf{b} =< b_1, b_2, \ldots, b_n >, \mathbf{y} =< y_1, y_2, \ldots, y_n >$$

To help us keep track of the forward propagation operations for each y_n, we will rewrite $\mathbf{y}$ as a vector defined by its functions.

$$\mathbf{z} =< (a_1 * w_1 + b_1), (a_1 * w_2 + b_2), \ldots, (a_1 * w_n + b_n) >$$

$$\mathbf{y} =< \sigma(z_1), \sigma(z_2), \ldots, \sigma(z_n) >$$

Next, we will define our desired output as the vector, $\hat{\mathbf{y}} =< \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n >$. We can use these vectors to compute the cost function. With the simple network, our cost function was defined on real numbers. However, in this case we can use the same function but with vector operations. The cost function will then return a vector rather than a single value. Since each weight contributes to the output of a single node, the vector allows us to do the backpropagation algorithm on n operations rather than just one.

$$C(\mathbf{w}) = (\mathbf{y} - \hat{\mathbf{y}})^2$$
$$C(\mathbf{w}) = \left\langle (y_1 - \hat{y}_1)^2, (y_2 - \hat{y}_2)^2, \dots, (y_n - \hat{y}_n)^2 \right\rangle$$

This cost function tells us how close each output values y_n are to the desired output values $\hat{y_n}$. We can evaluate the overall accuracy of an output vector by calculating the Mean Squared Error function, as defined below.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$$
$$MSE = \frac{(y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + \cdots + (y_n - \hat{y}_n)^2}{n}$$

The MSE is a useful metric for determining the loss of a network with n output nodes. We will come back to this when we implement the backpropagation algorithm in TensorFlow.

Now that we have the cost function, we just need to update the weights. We can do this by computing n partial derivatives, one for each weight for the fixed input a_1.

$$\nabla C = \left\langle \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \dots, \frac{\partial C}{\partial w_n} \right\rangle$$

This vector is known as the gradient vector denoted as ∇C, where each $\frac{\partial C}{\partial w_n}$ is the same partial derivative from the previous examples. To calculate the new weights, we multiply ∇C by the learning rate r, and subtract from the old weights to get $\hat{\mathbf{w}} = \mathbf{w} - r\nabla C$. Thus,

$$\hat{\mathbf{w}} = \left\langle w_1 - r\frac{\partial C}{\partial w_1}, w_2 - r\frac{\partial C}{\partial w_2}, \dots, w_n - r\frac{\partial C}{\partial w_n} \right\rangle$$

Once again we could iterate this until we converge on a set of weights that produce the desired output. Additionally, the same process is done for the biases. We will exclude these calculations since they are relatively the same. This is all need for the row operations of a fully connected network, so let us move on to the column operations.

Consider the new single layer network with n input nodes a_n, and a single output node y_0.

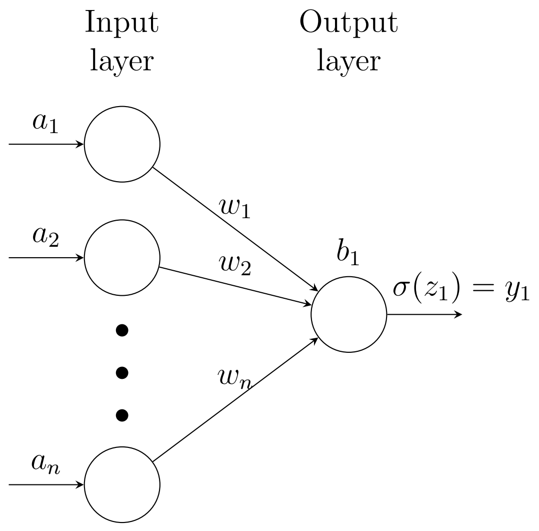


Figure 14: Graphic altered from Mark Wibrow's code [2]

To compute y_1, let us define our vectors for the inputs and weights.

$$\mathbf{a} =< a_1, a_2, \ldots, a_n >, \mathbf{w} =< w_1, w_2, \ldots, w_n >$$

Just like the matrix multiplication for our MLP, we multiply the input by the weights and add the bias, $\mathbf{a} * \mathbf{w}^T + \mathbf{b} = z_0$. Thus,

$$z_0 = (a_1 * w_1 + a_2 * w_2 + \cdots + a_n * w_n) + b_1$$

Finally, we apply the activation function to get $y_0 = \sigma(z_0)$. Now that we have propagated forward, we can implement the backpropagation algorithm, where $\hat{y}_1$ is the desired output. Again, we'll start by calculating the cost function, and then take the partial derivatives with respect to weights for each input.

$$C(\mathbf{w}) = (y_1 - \hat{y}_1)^2$$
$$\nabla C(a_1, a_2, \ldots, a_n) = \left\langle \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n} \right\rangle$$

This time, the partial derivative of the cost function is still a gradient vector ∇C because we are taking the derivative for each input a_n, with respect to the corresponding weight w_n collected as a vector. We must consider every input with every weight. Since each weight contributes to y_1 in some small way, we have to compute each partial derivative separately. Once we do this, the resulting gradient ∇C can be used to update the weights with the learning rate r, denoted by $\hat{\mathbf{w}} = \mathbf{w} - \nabla C r$.

Using both of these gradient vectors, we can now piece together the gradient matrix needed to update the weights in a fully connected network (one layer in the MLP).

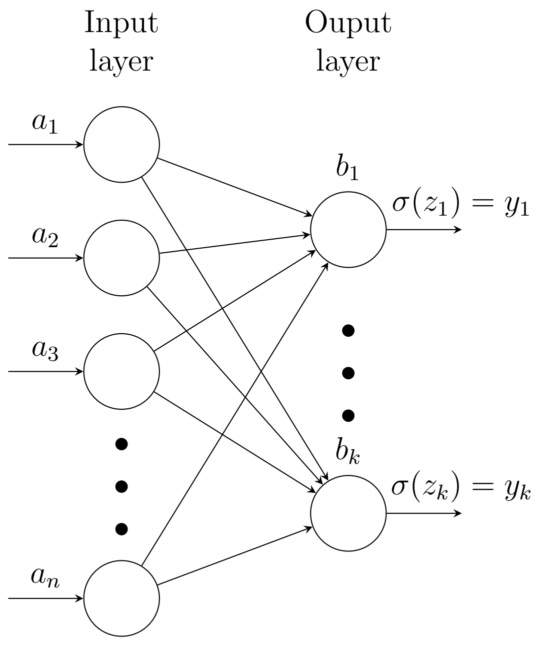


Figure 15: Graphic altered from Mark Wibrow's code [2]

We will start by writing our inputs and outputs as vectors, and the weights as a matrix. The indices of the weights are denoted by their layer connection as w_{kn}, where k are the output nodes and n are the input nodes.

$$\mathbf{a} =< a_1, a_2, \ldots, a_n >, \mathbf{y} =< \sigma(z_1), \sigma(z_2), \ldots, \sigma(z_n) >$$

$$\mathbf{w} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nk} \end{bmatrix}$$

Then, we propagate forward the input using matrix multiplication resulting in the following output from the equation $\mathbf{y} = \sigma(\mathbf{w} * \mathbf{a}^T + \mathbf{b})$.

$$\mathbf{z} = \begin{bmatrix} a_1 * w_{11} + a_1 * w_{12} + \cdots + a_n * w_{1n} + b_1 \\ a_1 * w_{21} + a_2 * w_{22} + \cdots + a_n * w_{2n} + b_2 \\ \ddots \\ a_1 * w_{k1} + a_2 * w_{k2} + \cdots + a_n * w_{kn} + b_k \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$$

$$\mathbf{y} =< \sigma(z_1), \sigma(z_2), \ldots, \sigma(z_k) >$$

Now that we have defined the output, we can apply the backpropgation and solve for the gradient matrix $\nabla C_{\mathbf{w}}$, where $\hat{\mathbf{y}}$ is the desired output. We start by calculating the cost function as follows,

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

Hence, we obtain the following gradient matrix by combining the two operations we defined before. For each row we find the gradient for that weight with its associated input a_n, resulting in n partial derivatives for each w_k in the respective row k. Then we do this for each column k times resulting in the gradient matrix $\nabla C_{\mathbf{w}}$.

$$\nabla C_{\mathbf{w}} = \begin{bmatrix} \frac{\partial C_1}{\partial w_{11}} & \frac{\partial C_1}{\partial w_{12}} & \cdots & \frac{\partial C_1}{\partial w_{1k}} \\ \frac{\partial C_2}{\partial w_{21}} & \frac{\partial C_2}{\partial w_{22}} & \cdots & \frac{\partial C_2}{\partial w_{2k}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C_k}{\partial w_{n1}} & \frac{\partial C_k}{\partial w_{n2}} & \cdots & \frac{\partial C_k}{\partial w_{nk}} \end{bmatrix}$$

Note that the bias gradient vector $\nabla C_{\mathbf{b}}$ would be calculated similarly. We can now use these gradients $\nabla C_{\mathbf{w}}$ and $\nabla C_{\mathbf{b}}$ to update the weights and biases using the learning rate r, by the equations $\hat{\mathbf{w}} = \mathbf{w} - \nabla C_{\mathbf{w}} r$ and $\hat{\mathbf{b}} = \mathbf{b} - \nabla C_{\mathbf{b}} r$. Thus resulting in the backprogation for a single layer fully connected network.

To extend this example for a multi-layered network, we would compute these gradient vectors for each set of weights $\mathbf{w}^L$ and biases $\mathbf{b}^L$ indicated for L number of layers. We would apply the same process as shown in the multi-layered network section (insert link), to each of the weights and biases propagating the partial derivative backward. Once we have L gradient matrices for each set of weights and biases, we would iterate the weight updates until we converged towards the desired output $\hat{\mathbf{y}}$. Moreover, as we will see in the next section, these iterative calculations of the partial derivatives for each layer are easy to produce.

Network Training

Now we can apply our understanding of the backpropagation algorithm to training our Multi-Layer Perceptron for recognizing handwritten digits. To implement this, we need to iterate over the 60,000 training images and then evaluate the model's accuracy with the 10,000 testing images. However, for each example in the backpropagation section (see pg. 37), we only looked at updating the weights and biases for a single input, implying that after training, the network would only be able to output the correct prediction for that input. We cannot do this for the MLP since we want it to classify ten possible outputs. The solution is to average the gradient matrix for an arbitrary layer L, $\nabla C_{\mathbf{w}^L}$ for multiple inputs to influence a single weight update. This is usual done in batches during the training phase, meaning we only update the weights and biases after we take the average of n inputs, where $\nabla \hat{C}_{\mathbf{w}^L} = \frac{1}{n}\sum_{n=1}^{n} \nabla C_{\mathbf{w}^L}$ (and similarly for the bias). Then for the batch of size n we update the weights and biases with the averaged gradient matrix as $\hat{\mathbf{w}} = \mathbf{w} - \hat{C}_{\mathbf{w}^L} r$ and $\hat{\mathbf{b}} = \mathbf{b} - \nabla \hat{C}_{\mathbf{b}^L} r$. A single iteration of these batch updates to the model for all training inputs is known as an epoch. Since we are taking an averaged gradient, a single epoch is insufficient for training an optimal network. Choosing the number of epochs depends on the learning rate, the data set, and the optimization algorithm. Once we have trained for any number of epochs, we can then evaluate the model.

We will start by splitting our training data into batches using TensorFlow's **Dataset** module.

BATCH DATA PREP
```
batch_size = 10
train_dataset = tf.data.Dataset.from_tensor_slices((train, lbtrain))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)
```

For now, we will use a batch size of ten with the idea that each input is one of the possible ten digits. Then we create a data set pairing the training images to their labels. Lastly, we shuffle the data set to randomize the batches for the first training loop. The function *shuffle()* takes a single argument called the buffer size. This buffer size is used to create the newly shuffled data set by replacing each data point in the buffer with another random point. The function then iterates this shuffling of the buffer until the whole data set has been covered. It is important to shuffle the data, so the images are not in any particular order when training. Now that the data is split into batches let us define our training loop.

TRAINING LOOP

```
def training_loop(model, train_dataset, learning_rate=0.9):
    epochs = 10
    for epoch in range(epochs):
        print("\nStart of epoch %d" % (epoch,))
        for batch, (x_batch_train, y_batch_train) in enumerate(train_dataset):
            train(model, x_batch_train, y_batch_train, epoch, learning_rate)
```

This function takes three arguments, the model itself, the batched training data, and the learning rate, which is 0.9 by default. Then we set the number of epochs (ten for now), and finally, we create the training loop. It starts by iterating over each epoch and then looping over the training data, then the **for** loop enumerates over the training data so we can eventually record the batch number we are currently on while also collecting the batch of images and their respective labels. Lastly, we then pass to the function *train*, the model, the batches, the number of epochs, and the learning rate.

The *train* function is where we will implement the backpropagation algorithm for each batch. As discussed earlier in the section, we need to compute each image's cost function in the batch and then average the gradients. However, the cost function only calculates the accuracy and gradient for a single input. To find the cost for each image and take the average, we can use the Mean Squared Error function for n inputs.

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$$

$$MSE = \frac{(y_0 - \hat{y}_0)^2 + (y_1 - \hat{y}_1)^2 + \cdots + (y_n - \hat{y}_n)^2}{n}$$

The MSE function is also known as a ***loss function***. In general, the output of a loss function is a metric for determining a model's accuracy for a set of inputs. In this case, due to the sigmoid activation function, loss values closer to zero means the model is producing accurate outputs, whereas values closer to one indicates that the model is performing poorly. When training, we can track each batch's loss function and find the average loss for the entire epoch. This will gives a better understanding of the model changes in performance over time.

Since the MSE function is still differentiable with respect to the weights and biases (summing the cost and dividing by n does chance the differentiability of the function), we can calculate the gradient matrices using this loss function in a single pass of the batch.

To implement the MSE function in TensorFlow, we can leverage the Tensor operations to calculate the squared difference between the entire batch and their respective labels. We can then average these costs by using the function *reduce_mean()*, which will return the loss value.

LOSS

```
def loss(predicted_y, target_y):
  return tf.reduce_mean(tf.square(predicted_y - target_y))
```

With this loss function, we are ready to implement *train()* which is shown below.

TRAINING

```
def train(model, train, lbtrain, epoch, learning_rate):

    with tf.GradientTape() as tape:

        current_loss = loss(lbtrain, model(train))

        dw1, db1, dw2, db2 = tape.gradient(current_loss,
        [model.layer1.weights, model.layer1.biases,
        model.layer2.weights, model.layer2.biases])
        model.layer1.weights.assign_sub(learning_rate * dw1)
        model.layer1.biases.assign_sub(learning_rate * db1)

        model.layer2.weights.assign_sub(learning_rate * dw2)
        model.layer2.biases.assign_sub(learning_rate * db2)
```

The function takes five arguments, the model, the training data (which is a batch of images), their respective labels (one for each image in the batch), the current epoch, and the learning rate. We start by defining the *tape* we will use for recording the operations since Tensor operations defined in the **with** block are automatically recorded on the *tape*, we must feed our batch of images to the model inside this block. Here, we do this by giving the model the entire batch, which returns each output for the batch in a single Tensor. Since the outputs are associated with the correct labels, we can give the models output and those labels to the loss function, which returns the loss value.

Remember, all of these operations performed are recorded in order onto the *tape*. Unfortunately, we do not see this happening, and it is obvious how this is done. However, we know that the *tape* can record these operations because we made sure that our model was well defined on the computational graph. That being the case, we can just let the API record these operations automatically to be used when performing the backpropagation algorithm.

This brings us to the next line of code which does all of the math we discussed in the *Backpropagation* section, where we calculated the partial derivatives used to update the weights and biases. We saw with the simple network that the function, *tape.gradient()* could take any number of partial derivatives for a given function. Our goal with the *train* function is to update the weights and biases using the MSE function through gradient descent. To do this, we tell TF to calculate these gradients by giving the function *tape.gradient()*, the loss output *current_loss*, and a list of the **Variables** we wish to take the partial derivative with respect to.

Here, the gradient matrices (dw1, db1, dw2, db2) are calculated using the same partial derivatives we defined in the previous section ($\nabla C_{\mathbf{w}^1}$, $\nabla C_{\mathbf{b}^1}$, $\nabla C_{\mathbf{w}^2}$, $\nabla C_{\mathbf{b}^2}$). The function *tape.gradient()* does this through the symbolic differentiation, which iterates over the *tape* backwards and calculates the correct gradients with respect to each defined variable. So the output of this function gives us our averaged gradients of the partial derivatives for the layer one weights (dw1) and biases (db1), and the layer two weights (dw2) and biases (db2).

Then, we can use these partial derivatives to update each set of weights and biases. The last four lines do these updates by multiplying the gradients by the learning rate and then subtracting the gradient from the original set of variables. Once the weights and biases are updated in the model, the process is then repeated for all training samples and epochs.

This leaves us at the finale of our work, training and evaluating the model. Before we call our train function, we need to have a responsive test function that iterates over the testing images and records the network's overall accuracy. In the future, we will explore other evaluation methods, but here we want to see the Multi-Layer Perception's accuracy for the 10,000 testing images. We can implement this by iterating through each image, feeding it to the network, and recording the network's output as either correct or incorrect. Then we can take the total number of images correctly labeled and display the model's accuracy.

TESTING

```
def testing_loop(model, test, lbtest):
    a = tf.keras.metrics.categorical_accuracy(model(test), lbtest)
    correct = float(tf.math.count_nonzero(a))
    total = float(len(test))
    accuracy = (correct/total) * 100.0
    print("CORRECT: ", correct, "ACCURACY: ", accuracy)
```

The testing function takes three arguments, the model, the testing images, and their labels. We start by calling the function provided by **Keras**, *categorical_accuracy()*, which takes two arguments: the prediction outputs and the actual (or desired) outputs. This function evaluates the accuracy of *hot-encoded variables*. These variables are labels turned into a Tensor where the position of a one indicates that label (and the rest of the values are zero). We already did this when prepossessing the data to make the backpropagation possible. This encoding is widely used in neural networks, and this function makes it easy to record the accuracy of outputs that are hot-encoded and only works if the prediction outputs are values between zero and one. The function then returns a Tensor containing an encoding of the accuracy, where 1 is a correct output, and 0 is an incorrect output. It does this by finding the max value in the prediction output and comparing its position in the Tensor to the actual output. If the actual output has a one in that position, then the function records 1. Since we are dealing with Tensors, we can use this function to give us the accuracy encoding for every output and their respective labels. Since the values in this output Tensor called, **a**, are either 1 or 0, we can then count the number of correct values by using the next function *count_nonzero()*. Finally, we calculate the total, and then we can get the accuracy.

Now that we have all of the pieces to our Multi-Layer Perceptron, let us call these two functions and see how our network performs. Below is the output of calling the training and testing loops with 10 epochs and a learning rate of 0.9.

NETWORK ACCURACY OUTPUT

```
Start of epoch 8

Start of epoch 9
CORRECT:  9081.0 ACCURACY:  90.81
```

After only 10 epochs, we have an accuracy of about 91%, which is where the real fun begins. There are so many different ways to increase this accuracy by increasing the number of epochs, changing the learning rate, or even altering the network's number of hidden layers. But even with a relatively small model, we were able to teach the neural network how to recognize handwritten digits with respectable accuracy.

Network Evaluation

We have finally built our MLP that can recognize handwritten digits. However, there are still many lingering questions about our model. What happens if we use different learning rates, change the number of hidden nodes, and how do these variables affect the model's performance? This section will be devoted to answering these questions and more; through empirically testing multiple variations of the MLP. So far, we have only looked at the accuracy of one trained MLP. However, this single result is not a measurable representation of how the model would perform with real-world data sets. We can evaluate our models' performance holistically by using the model validation method known as *cross-validation*. This method splits the input data into multiple sections to iteratively train and test the model on each section of the data, ensuring that the model's accuracy is not biased towards a specific training and testing data split.

In particular, we will implement the model validation known as *K-Fold Cross Validation*. This method splits the data into k sections (or folds), which are randomized. Then the model evaluates the accuracy and error k times, where each iteration uses a single fold as the testing data and the remaining folds as the training data. We then calculate the model's overall performance by averaging each k-fold's accuracy and error. Below is helpful visualization of the K-Fold Cross Validation algorithm.

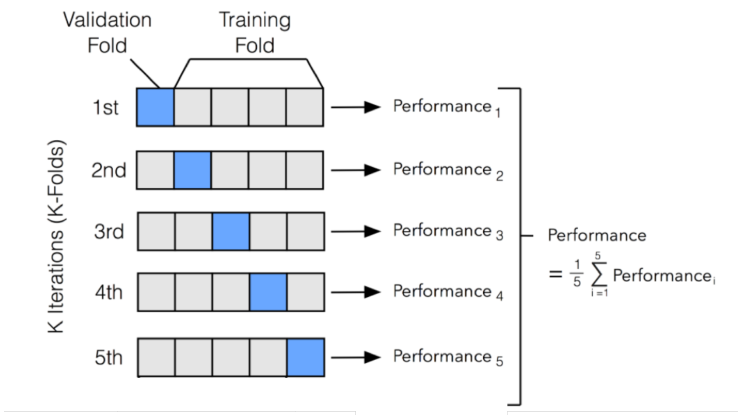


Figure 16: Image borrowed from scikit-learn [3]

It is important to note that Cross-Validation can run into issues when attempting to evaluate classification models. In particular, this method does not work well with imbalanced data, which happens when there is a significant difference between the number of samples given per class. Thus, when we split the data into the k-folds, a fold can leave out an entire class altogether or only include one class. An example of this could mean that a specific fold would only contain the images of a 1 and 9 and leave out the rest of our model's images. The solution to this potential imbalance is not to randomize the entire data set at each fold but rather to randomize each training and testing split. Even when the original data is not imbalanced, we can deploy this kind of randomization to ensure that the training and testing splits do not leave any classes. The MNIST data set is well balanced, and this issue should not cause any misleading results. We will keep this in mind when implementing the K-Fold Cross Validation algorithm.

Unfortunately, TensorFlow does not have a straightforward implementation of this model validation method, so we have to implement it by hand. Additionally, to make it easier to test, we will also parameterize the MLP model, which requires completely refactoring the code we have written so far. Most of these changes are self-explanatory, so we will not cover the entire implementation. The complete code exists in Appendix B linking to a Github, which provides helpful comments on those changes.

After the refactoring, we are now able to call the method, *evaluate()*, which takes seven arguments, the model, the data set, the labels, and the variables we wish to test: the learning rate, the number of epochs, batches, and the number of splits used in cross-validation (Appx. B, code line 119). It is important to note that the method *evaluate()* will only run K-Fold Cross Validation if the value of k is valid. Otherwise, it uses our original data split of 60,000 training and 10,000 testing images with one iteration. The value of k will only be valid if it is greater than one, less than the number of data samples, and evenly divides our data set. This last qualification makes the implementation much more straightforward because we do not have to worry about the remaining data samples.

The most significant change to our network is how we define the network's architecture. Before, we assumed that the network had a single hidden layer. Therefore, we only used two Perceptrons, one for each layer. However, to allow for more layers to be added (Appx. B, code line 33), we have generalized the layer creation method. To define our network, we pass in a list corresponding to the number of nodes we want at each layer. For example, if we wanted to build the same network that we already have, which has 784 input nodes, 15 hidden nodes, and 10 output nodes, we pass our MLP constructor the list: $[784, 15, 10]$. Now we can easily change the number of hidden layers and how many nodes are at each layer.

In summary, by refactoring the code, we have made it easier to test with and without cross-validation. However, it is unnecessary to use this code to answer our questions about the MLP since we show the results in the following sections.

Cross-Validation Comparison

We will start by comparing the results of a single models' accuracy to the accuracy determined by cross-validation. Additionally, we will compare three separate networks with different learning rates to see how the learning rate affects the model performance. For this test, we use a 5-fold validation split to give us the best result without sacrificing computation time. The trained MLPs use ten epochs, a batch size of ten, and have the network structure [784,15,10]. Below are the results for the accuracies and variances from each set of training methods using three learning rates.

Learning Rate	Accuracy	Error	Total Run Time
0.9	89.12	0.01749	91.21s
0.1	58.42	0.05698	96.47s
0.01	16.51	0.093112	92.64s

Figure 17: No validation

Learning Rate	Accuracy	Error	Variance	Total Run Time
0.9	90.81	0.012977	1.91	426.27s
0.1	79.63	0.030024	101.18	433.25s
0.01	36.57	0.078705	67.89	438.14s

Figure 18: 5-fold validation

We can glean from the results that the overall accuracy for each learning rate with 5-fold validation (Fig. 18) is an improvement from taking the results from a single output (Fig. 17), which is an example of how much variation a model can have based on a single performance. Just because a model's performance is good or bad for a single training split does not determine how it will perform every time we train it. Many factors go into the poor results we see from the single validation. However, we can still analyze these issues by locating potential problems with the model.

In both evaluations, we can see a clear trend in the model's accuracy as we decrease the learning rate. We know from gradient descent that the learning rate determines how much we change the weights and biases to converge toward the minimum. As the learning rate gets small, the weight and biases will change by a smaller amount for each update, which translates to the error value being minimized slower than to a model with a greater learning rate, which results in a poor accuracy score.

Since we are using 5-iterations to represent our performance with cross-validation, we must analyze the variance of each iteration's accuracy to evaluate the model consistency. Ideally, we would like to build a model with a low bias and variance, meaning our model should produce almost the same error value at each kth iteration. From these results, we can see that the model's variance with a learning rate of 0.9 is significantly less than the variances of models with learning rates 0.1 and 0.01. From these results alone, it is not at all clear why this might be the case.

To investigate our accuracy and variance disparities, we will attempt to improve our models' performance by increasing the number of epochs. Moreover, it should allow for gradient descent to go further in minimizing the error. Below are the results after increasing the number of epochs to 25.

Learning Rate	Accuracy	Error	Total Run Time
0.9	91.55	0.013766	213.71s
0.1	83.24	0.027203	219.58s
0.01	46.05	0.071884	211.93s

Figure 19: No validation

Learning Rate	Accuracy	Error	Variance	Total Run Time
0.9	93.01	0.009271	1.35	1055.59s
0.1	87.33	0.018668	9.06	1064.35s
0.01	50.27	0.063937	116.71	1068.03s

Figure 20: 5-fold validation

Now that our models have made more progress towards minimizing the loss function, it should be no surprise that we see an overall increase in accuracy for both results.

We see a significant rise in the accuracy for the model with a learning rate of 0.01 (Fig. 20) in comparison to results from Figure 18. By contrast, the model with a learning rate of 0.9 had only a slight increase in performance, and the change in variance from the model trained with only ten epochs (Fig. 18) was negligible. Since we are not guaranteed to find the global minimum using gradient descent, the algorithm will more likely reach a local minimum, causing the model to plateau in performance leading to over-training.

Our two most significant changes in variance are that of the models with learning rates 0.01 and 0.1. From these results, we can hypothesize that as gradient descent further minimizes the error, the more consistent the ending model becomes, the more our evidence points to the model's tendency to over-train and plateau in performance. We can visualize this effect when we plot the error values after each epoch using the same test from our results in Figure 20.

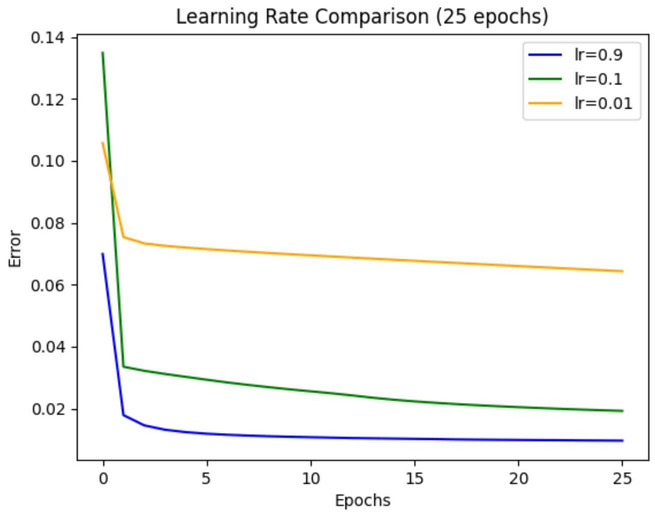


Figure 21: Error Trends

From the graph, we see the learning rates of 0.1 and 0.01 are still trending down. If we want these models to increase in performance, then we must increase the number of epochs. However, this will require more computation time for a similar result from the network with a higher learning rate. Therefore it might not be practical to use such a model since we can achieve its accuracy more efficiently. Furthermore, we see that the model with a learning rate of 0.9 reaches an error value of 0.01 much quicker than the others. However, over time this score plateaus with more epochs, consistent with the results from our change in accuracy seen in Figure 20.

There is an extreme case where training the model on an egregious number of epochs can cause the error to increase as it is over-trained. We can avoid this problem using an error threshold that cuts off the training for a given epoch if the model's error has reached the threshold. This solution is known as early stopping and often used when validating a model. In our case, it is unnecessary to implement this because we do not need to increase the number of epochs beyond a point where this becomes problematic since we have shown that our model performs well with a reasonable number of epochs and a large enough learning rate.

Decreasing Computation Time

We see that it takes roughly 18 minutes to run cross-validation with 5 folds on our model with 25 epochs from our previous results. It takes about 3.5 minutes for a model to be trained and tested. The simplest solution for faster computation is to use a GPU. Due to Tensorflows' limited support on only Nvidia GPUs, we will not show how to integrate GPU computation into our model. However, we can leverage Tensors and the computational graph to optimize the operations we have already defined.

We can use *tf.function* to increase the computation time of our Tensor operations. By default, TensorFlow runs in eager execution mode, which immediately evaluates the operations as though it was Python code. However, by applying *tf.function* as a decorator, we tell TensorFlow to optimize these operations by collecting all the operations in the computational graph first. Then once it has optimized those operations(having unnecessary operations or computing the same gradients multiple times), it will compute the model as we defined. Below is an example of how we indicate the use of *tf.function*.

TF FUNCTION

```
@tf.function
def __call__(self, x):
        return tf.matmul(x, self.weights) + self.biases
```

It is important to note that this decorator only works if the method uses Tensors and Tensor operations alone. We cannot use this decorator everywhere because it turns our Python function into the TF object called **Function**. **Functions** can be faster than eager code, especially for graphs with many small operations. The TF **Function** uses an algorithm known as tracing to collect all the functions operations inside a TF graph. Once these operations are collected in the graph, TF can decide how the operations should compute the graph's given structure reflected by our Python code. Performing tracing on non-Tensors can cause side effects such as object mutation or list appends, making our computation time even slower.

Luckily, we do not need to manually trace all of our Tensor operations since we do not need certain operations to rely on each other for the model to work at peak efficiency. We know that our most computationally expensive operations are computing the output and calculating the gradients. Since these procedures consist of many small operations, we can use *tf.function* as a decorator to our gradient descent function, the loss function, and the callable function for computing the output of our network. These changes are pointed out in the code sample from the Appendix B. It is clear that we did not use this decorator in the previous section, but as we continue to test our model, we should see a significant increase in our computation time.

Model Bias

In the previous testing, we saw that the model could be biased towards the training process itself. This section will look at how the model can become bias towards a given set of weights and biases in training, which can happen if the MLP has too many hidden layer nodes. To explore this, we will test a few MLP's of varying hidden layer sizes, all with a learning rate of 0.9, 10 epochs, and a batch size of 10 using 5-fold validation.

Model	Accuracy	Error	Variance	Total Run Time
[784,15,10]	90.92	0.012391	2.37	101.10s
[784,50,10]	89.18	0.013969	2.94	113.37s
[784,100,10]	94.33	0.008337	2.54	126.72s
[784,800,10]	44.22	0.059228	26.21	213.48s

Figure 22: Results from varying hidden layer nodes

First, we can note that the overall computation time has significantly decreased compared to our times in Figure 18. Secondly, we can glean from these results that increasing the number of hidden layer nodes from 15 to 50 did not change the network's overall accuracy. Yes, the accuracy for the 50 hidden node model did fall. However, it did not decrease enough for there to a distinguishable difference. 5-fold cross-validation is not a solution to training bias, and this decrease could be because of the model's bias towards its given training split.

We see a more considerable change to accuracy with 100 and 800 hidden nodes. In this test, it is apparent that 100 hidden nodes have the best overall performance so far. It is not easy to reason why exactly 100 nodes caused such a significant increase in accuracy, and more specific analysis is necessary to come to any conclusions. It is even possible that if we increased the number of epochs, this accuracy would be even better. These potential issues show how much work must go into building a model for solving a particular problem. The given variables to a model will not always be optimal, and it takes more testing to determine how those variables should be changed to achieve the best performance.

Finally, the outlier in these results is the model with 800 hidden layer nodes, which is the situation we mentioned earlier. We can see a significant drop in performance when we use too many hidden layer nodes. Plus, the model was more inconsistent for every 5 iterations of the cross-validation. This performance drop does not happen with the other models and points to the issue of the MLP becoming biases towards a given set of weights and biases. We can analyze the specific digits that the model correctly predicts on all 70,000 images after training and testing with 5-fold cross-validation to investigate this issue.

Labels	0	1	2	3	4	5	6	7	8	9
Number of Correct Predictions	6807	7726	30	6829	3	5959	2	7092	6489	6607

Figure 23: Model: [784, 800, 15]

This table shows why the model with 800 hidden nodes performed poorly and inconsistently. By adding more hidden nodes than inputs, the model can memorize a particular set of classes. Then once the model has those images memorized, it will only be able to predict those classes. For example, with this model, we see that it rarely correctly predicted the digits 2, 4, and 6, which is evidence that the model memorized other digits and never considered predicting these digits with low prediction scores. This problem is not evident in our other models, as shown below for the model with 100 hidden nodes.

Labels	0	1	2	3	4	5	6	7	8	9
Number of Correct Predictions	6652	7574	6369	6524	6494	5621	6584	6879	6122	6237

Figure 24: Model: [784, 100, 15]

Since this model was 94.33 percent accurate, we are not surprised that it almost correctly predicted every image. We also see an even distribution of these predictions, which implies that our model does not favor a given class and would have consistent predictions. These results follow the previous testing we have done and lead us to conclude that it is possible to create an efficient and accurate model with one hidden layer given the correct parameters. This statement might be an unsatisfactory answer, but the truth behind building the most optimal network comes down to optimizing the parameters to achieve the best results. Our goal was not to find the best MLP for our problem but rather to explore the process of evaluating and fine-tuning a model. After all, we have not done enough analysis to conclude the most optimal model for predicting handwritten digits.

Multiple Hidden Layers

All this work has led us to the final section, where we get to test MLPs with multiple hidden layers, which is a type of deep neural network (DNN). Other DNNs include the convolutional neural network, which has become increasingly popular and easily implemented in TensorFlow! DNNs are the networks that play chess, drive cars, and play video games. It is important to note that solving these problems using just an MLP with multiple hidden nodes fell out of fashion due to the difficulty in training these networks. Some of these networks would require more than just three hidden layers, and with so many variables, it becomes difficult to figure out which weights and biases give the desired output. We cannot expect adding hidden layers correlates with better predictions. Adding these layers does allow the network to solve more complicated problems, but just as we showed in the AND/OR to XOR problem, there are tangible reasons behind the type of network we need to solve a given problem.

However, this is just a disclaimer for building these types of networks, and as we have shown, it takes a significant amount of evaluation and analysis to determine the best network for the job.

Yet, we can still have some fun by building and testing a variety of MLP hidden layer structures.

Name	Epochs	Lr	Model	Accuracy	Error	Variance	Run Time
1	10	0.9	[784, 30, 15, 10]	92.95	0.009329	4.05	119.23s
2	25	0.9	[784, 30, 15, 10]	94.53	0.006505	2.13	307.86s
3	25	0.9	[784, 15, 30, 10]	93.9	0.007328	1.83	291.17s
4	25	0.9	[784, 1, 100, 10]	36.61	0.070558	23.11	270.96s
5	25	0.9	[784, 100, 50, 10]	96.09	0.004035	2.83	392.57s
6	25	0.9	[784, 50, 100, 10]	96.16	0.004302	2.2	343.72s

Figure 25: Two Hidden Layers

We start our adventure by testing a swath of MLPs with two hidden layers. We see from models 1 and 2 that it is more accurate with more epochs; go big or go home! Unfortunately, we do not see a significant increase in performance with these two networks than the models with one hidden layer. However, models 5 and 6 do increase in accuracy by 2 percent. From these results, we can infer that networks with more hidden nodes at each layer allow the network to extract more features from the inputs (we also saw this with the XOR problem to the MNIST). However, this is likely dependent on how many hidden nodes are at each layer. This idea is tested with models 2 and 3 and with 5 and 6. In both, we reverse the number of hidden nodes at each layer, and from those results, it is too difficult to tell if this makes a significant difference. Which leads us to ask the question: what happens when the number of hidden nodes increases or decreases at each layer, and does it affect the performance?

We take this question to the extreme with model 4 as it is fundamentally flawed. When the inputs go to a single output, the network can only describe the images from the weights and biases contributing to the single output. This limitation points to a potential problem for networks with an increasing number of hidden nodes from one layer to another. Let us keep this in mind as we analyze the results from the networks with three hidden layers.

Name	Epochs	Lr	Model	Accuracy	Error	Variance	Run Time
7	25	0.9	[784, 200, 100, 50, 10]	96.78	0.002594	4.47	508.08s
8	50	0.1	[784, 200, 100, 50, 10]	93.61	0.00775	6.54	1048.57s
9	25	0.9	[784, 100, 50, 10, 10]	96.04	0.003728	3.46	392.53s
10	50	0.7	[784, 100, 50, 10, 10]	96.48	0.002793	3.41	824.14s
11	25	0.9	[784, 10, 50, 100, 10]	92.86	0.008865	1.74	316.39s

Figure 26: Three Hidden Layers

Models 7 and 8 test the learning rate, and we see a similar difference in performance compared to our first tests (Fig. 18 and Fig. 20). These learning rate tests are continued with models 9 and 10 and are consistent with previous results.

However, we see a significant change from model 9 to model 11, where we reverse the number of hidden nodes. Model 11 is less accurate, and we can reason why. We know that the MLP tries to extract input features. Since model 11 sends all 784 to only 10 outputs nodes in the first layer, it only extracts 10 features. Even though we only have ten digits, this still over-generalizes the potential distinctions between the same digit. Then, those generalized features are sent to another layer with more outputs, forcing the network to describe the 10 features as 50 features, and finally as 100 features. Hence an increase in the number of the hidden nodes from layer to layer seems like an inefficient network structure for more extensive networks.

With our last tests, we take this problem to the extreme for networks with 5 hidden layers (all of the models use 25 epochs and have a learning rate of 0.9).

Name	Model	Accuracy	Error	Variance	Run Time
12	[784, 400, 200, 100, 50, 25, 10]	96.89	0.00193	6.79	787.62s
13	[784, 25, 50, 100, 200, 400, 10]	71.05	0.032778	933.92	610.88s
14	[784, 200, 100, 400, 25, 50, 10]	96.54	0.002667	5.13	693.05s
15	[784, 10, 100, 400, 200, 50, 10]	92.84	0.008864	1.45	612.49s

Figure 27: Five Hidden Layers

Now it is clear that models with an increasing number of hidden nodes will perform poorly and inconsistently. Model 13 had the worst variance out of all of the tested models and is more evidence towards the answer to our original question about the number of hidden nodes from layer to layer.

In model 13, the first layer, 784 to 25, generalizes the inputs into 25 features. These 25 features go to more extensive layers in the subsequent layers that try to extract information from our generalized features. This problem does not persist in model 14 as the first layer contains 200 features which more than enough to perform well. However, in model 15, the accuracy decreases after changing the number of nodes in the first hidden layer. Surprisingly, this change from model 14 to 15 does not worsen the score significantly beyond model 11.

We can conclude from these results that an increasing number of hidden nodes per layer leads to good performance, where a decreasing number of nodes leads to inconsistent and poor performance. It is not apparent how networks with varying hidden layer sizes should improve performance. However, we can see how the number of nodes in the first layer does change the network's accuracy, most likely caused by the generalization made in the feature extraction at the beginning of the network.

Thus, the number of nodes in the first layer is an important variable to consider towards extracting enough features in the inputs to perform well. For the subsequent layers, the same logic can be applied depending on the classification problem. In our case with handwritten digits, the images are not very complex nor contain noise, which means we do not need an ultra-complex network structure to solve our problem. It might not even be necessary to have more than one hidden layer. Although these larger MLPs are fascinating, the trade-off in computation time alone is enough to rule these networks' efficacy in real-world situations. At least now we know how we might structure a larger network for solving even more challenging problem than recognizing handwritten digits.

Conclusion

In summary, we started by setting out this goal to recognize handwritten digits using TensorFlow and Multi-Layer Perceptrons. Unfortunately, the TensorFlow API is esoteric and requires an established knowledge of building neural networks. To break down TensorFlow, we simplified our network to just a Perceptron. We showed how limited this network is for solving classification problems leading us to build the MLP. Along the way, we learned how to build networks in TensorFlow effectively. We mainly focused on matrix multiplication and ensuring that our network was comprehensible to the computational graph.

Once we were confident of our integration, we could abstract our model to build a Multi-Layer Perceptron for recognizing handwritten digits. However, as our network became larger than a Perceptron, the task of determining the correct set of weights and biases for producing a prediction became increasingly difficult. The solution was to implement gradient descent.

We plunged into the relevant multivariable calculus and linear algebra to teach the network how to recognize handwritten digits. These derivations did not come by easily and would be tedious to replicate with other functions. TensorFlow's library remedies this problem for calculating partial derivatives using Automatic Differentiation. It is a powerful tool allowing us to abstract the operations necessary for gradient descent, making the implementation simple to code.

After building an MLP that could recognize handwritten digits, we explored the hyper-parameters (learning rates, epochs, and hidden layers) of the MLP using K-Fold Cross-Validation. As we explored these testing results, we understood what it takes to build a network that could handle real-world data. It was necessary to parameterize our network and generalize the overall structure to implement the validation method. In refactoring, we ended up building a network class that closely resembles the higher-level class implemented by Keras.

The Keras library makes it very easy to build a network using larger data structures representing entire layers of the network. For example, to build a two-layer network, we would use a sequential model that contains a single **Dense** layer that is a Keras class, which contains all of the hyper-parameters necessary for building a fully connected layer. Keras truly makes it easier to build these larger networks, and after exploring this tutorial, most of the documentation relating to the neural networks should feel familiar.

With this foundation in TensorFlow, it should be much easier to build models that solve other problems. However, we are not limited to building Multi-Layer Perceptrons. Using Keras and TensorFlow, it is easy to build more popular networks such as Convolution Neural Networks (used for image classification) and Recurrent Neural Networks (used for text prediction). There are helpful tutorials on their website that should be easier to read now that we have a solid TensorFlow foundation, don't stop here!

Appendix A

Installing Packages

To install the packages, please have the latest version of Python installed, check for updates here, https://www.python.org/downloads/.

TensorFlow 2.0

We will be installing the latest version of TensorFlow 2.0. TensorFlow is tested and supported on the following 64-bit systems: Ubuntu 16.04 or later, Windows 7 or later, MacOS 10.12.6 (Sierra) or later (no GPU support), and Raspbian 9.0 or later.

Install TensorFlow

```
#The latest version of pip (>19.0) is required to use TensorFlow
$pip install --upgrade pip

# Install the current stable release for CPU and GPU
$pip install tensorflow
```

For more information on installation and building TensorFlow from source, go to: https://www.tensorflow.org/install.

Numpy

Install Numpy

```
pip install numpy
```

Matplotlib

```
Install Matplotlib
pip install -U matplotlib
```

Appendix B

Tutorial Code

The refactored code is not included here due to the size of the codebase and readability but can be found at the link, https://github.com/davidavzP/MLP-Tutorial-TensorFlow.

Bibliography

[1] Brett Szymik. Neuron anatomy. ASU School of Life Sciences Ask A Biologist, May 3, 2011. URL https://askabiologist.asu.edu/neuron-anatomy.

[2] Mark Wibrow. Multi-layer perceptron code. URL https://tex.stackexchange.com/questions/153957/drawing-neural-network-with-tikz.

[3] K-fold cross validation. URL http://ethen8181.github.io/machine-learning/model_selection/model_selection.html.