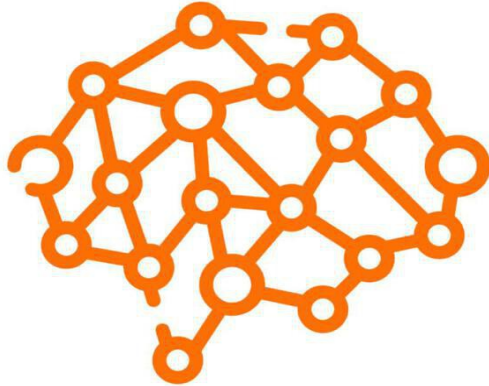


# Introduction to TensorFlow<sup>®</sup>

Using Python



Soam Desai

## My Note to You

Hey! Thanks for taking the time to read this wonderful book. My name is Soam Desai and I'm the author if it wasn't apparent from my name on the cover. This book is meant for the beginner who has some basic coding knowledge; however, if you don't have any coding experience, you can still tag along and learn a thing or two. This book guides you through the entire process. I still remember when I first started learning about TensorFlow and ai in general. It was tough and it still is. The concepts were abstract and you needed a deep understanding of multivariable calculus and Linear algebra to even get some of the concepts. I only achieved a greater understanding of the concepts after I took those classes. That will not be the case in this book. I have simplified the topics so that high schoolers, even ambitious middle schoolers can understand. We will focus mainly on the coding aspects of TensorFlow and I will make the concepts easier to understand with my quirky analogies and humor. With all that said, I hope you

enjoy and make sure to stick with it!

Note: \*TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc. 'Python premium icon: Flaticon.com'. The cover has been designed using resources from Flaticon.com.

## Table of Contents

Introduction.....  
.....4

Project 1: Image Classification of  
Clothing.....22

Project 2: Basic Text

Classification.....38

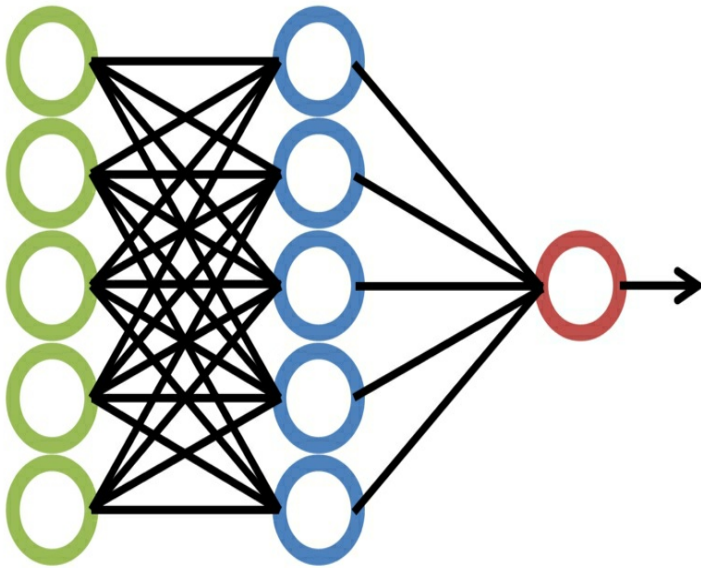
Project 3: Text Classification with  
TensorFlow Hub..62

Project 4: Predicting Fuel Efficiency.....  
.....72

Closing.....  
102

## Deep Learning

What is Deep learning? It's one of those buzzwords that's thrown around nowadays. Every new start-up in Silicon Valley seems to be doing something with "Deep Learning" or artificial intelligence. Well, deep learning is when a multi-layered neural network is trained with data so that the network eventually makes decisions that are similar to that of a human, or are even more optimal. If that sounds complicated, just read on. As seen below the network consists of many neurons which is where the phrase "neural network" comes from. Each neuron itself is simply just a function. It receives input and it sends output to the neuron after it.



I guess you could say scientists like to give stuff fancy names. The only reason why neural networks are called what they are is because the neurons in our brain perform a similar function (although our brains don't work with computer code). In the end, a neural network is just a bunch of interconnected functions; however, these functions can do some very powerful things.

Deep learning has been used by a variety of people to do some very interesting things. Some of the most notable of these are self-driving cars, facial recognition, composing

music, and much more! What makes deep learning so useful is that the network it is operating on can choose what input data is most useful. Instead of a human having to decide what input data to feed into the network, deep learning itself can recognize which inputs are useful and which are useless. Deep learning has already changed our world and will do so to a greater extent in our near future.

What is TensorFlow?



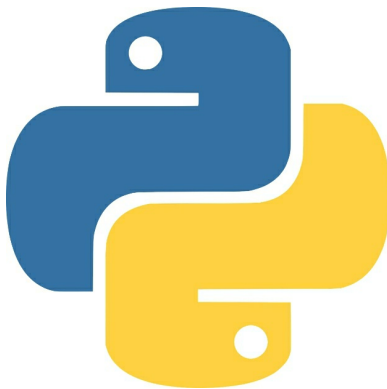
First off, because I included the



TensorFlow logo, I have to legally say that "TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc". Now that that's out of the way, let's talk about what TensorFlow is. The TensorFlow website defines TensorFlow as an "end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications". I like to think of it as a box in which we can use the tools inside to create something even better. Instead of having to reinvent the wheel, we're given the wheel and we're going to make a car! Although the website defined it as a platform, I'm just gonna refer to it as a library. Coding libraries are very common in programming world and all they are is just a few files of code that you can add to your existing code to do more tasks. As described above, this is what TensorFlow is at its very basic level. It gives us code we can use to write programs. TensorFlow has been used by many professionals around the globe and

is the most popular deep learning/machine learning library to date. It was invented by the Google Brain team in 2015 and its popularity has grown ever since. The library is available in many different programming languages, but we are going to be using the Python library because we're going to be using Python (I'll explain why later). Although we won't be doing anything cutting edge, we will be diving into the basics of TensorFlow and will definitely have some fun along the way. Just wait till you finish your first project.

Python, What's That?



Python is a type of constricting snake that is part of the Pythonidae family . . . oh wait, wrong type of Python. The type of Python we are actually looking for is the programming language. Wikipedia defines Python as, “Python is an [interpreted](#), [high-level](#) and [general-purpose programming language](#). Created by [Guido van Rossum](#) and first released in 1991”. All this really means to us is that Python can execute commands with less code and its easier to understand than other languages. Personally, Python is my favorite language because of how “easy” it is to read and write. It is also a great language for beginners which is why we will be using it in this book. Although Python is great and all, it does have some drawbacks, but these drawbacks will not affect us based on what we are going to do. The drawbacks are that Python is slower compared to other languages. Because it is very high level, i.e. easy to write and understand, it runs slower. It executes code line by line instead of compiling it and then running it like other languages. All you need to know is that Python is an awesome programming

language and I hope you use it more in your future! In this book, we will be more focused mainly on the deep learning aspect of Python which is why I recommend you further learn about the fundamentals of Python either through a different book, or through an online learning platform like Coursera. This is not necessary for this book, but I highly recommend you do.

## Setup

Now that you've gotten an introduction to all the tools and what deep learning is, we get to probably the worst part, setup. In the following pages, I have included a guide on what we're going to need to successfully run and create our cool projects. There is a much simpler way to setup that's pretty much already setup. This simpler way is Google Collab. Google Collab is an online code editor that is on the cloud. I highly recommend you use Google Collab. It includes everything we'll be needing including all the libraries. Personally, I will be using Google Collab; however, I have included this guide if you want to do it locally on your computer. If you want to skip to the Google Collab section, go to **page 18s**, otherwise read on.

The code for all the projects can be found at:

<https://drive.google.com/drive/folders/1zGtusp=sharing>

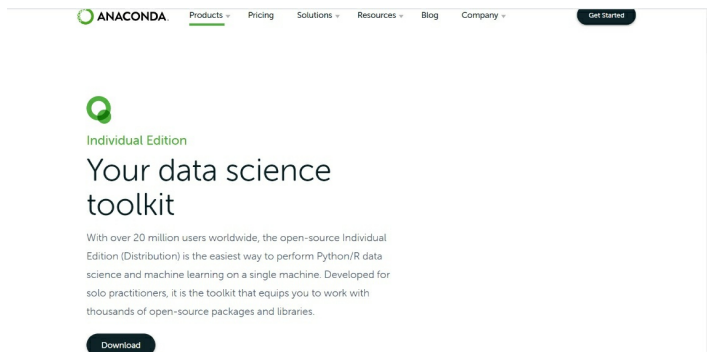
## Downloading Python

The first thing we need to do in our setup is to install Python. Instead of outright installing the normal installation of Python, we will be getting Anaconda instead. Don't worry, I'm not making another snake joke. Anaconda is a software that comes prepackaged with a lot of the libraries we will be using and it also comes with Python. Anaconda is widely used in the Data science/Machine learning communities which is why it's recommended.

First, go to this link:

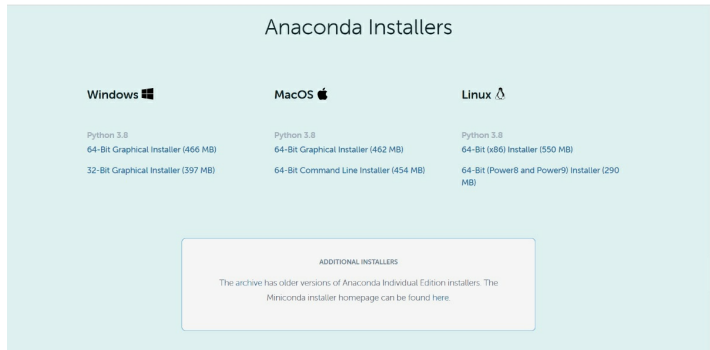
<https://www.anaconda.com/products/individual>

Then, scroll slightly down until you see the “Download” button:



Click the “Download” button and you will go to a screen that looks like this:



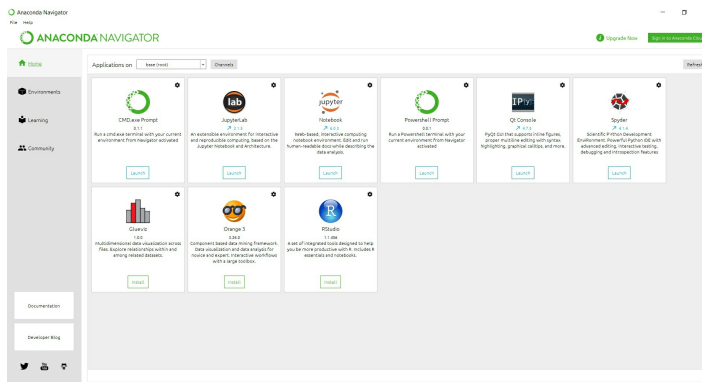


Click the Appropriate installer for your operating system (most likely 64 bit).

Then, follow through the installation steps. If you need help on this section, go to <https://docs.anaconda.com/anaconda/install/>

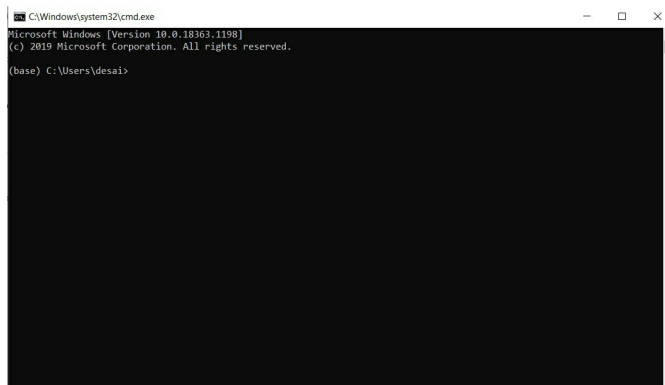
The installation process may take a while.

Once your installation of anaconda is complete, you can open the anaconda navigator to a screen like this:



Now that anaconda is installed, we need to download our required libraries

Begin by clicking launch on the CMD.exe Prompt. It's the icon in the top left. I have a windows pc, so if you have a mac, it might look slightly different. Once you do that, you should get a terminal screen



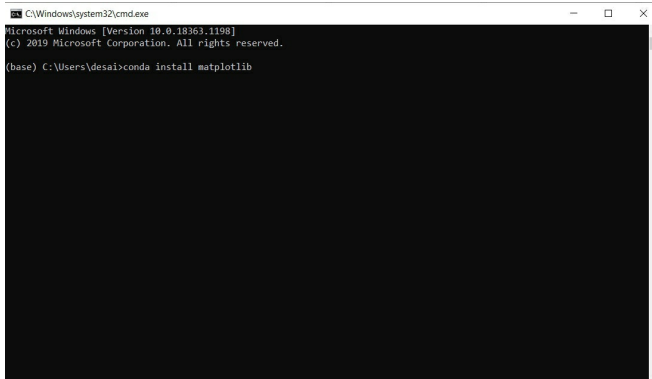
The libraries we need to download are:

1. pathlib
2. matplotlib
3. numpy
4. seaborn
5. Ipython
6. pandas

We can install the libraries one by one by typing:

“Conda install” followed by the package name. Make sure that the capitalization is there if needed.

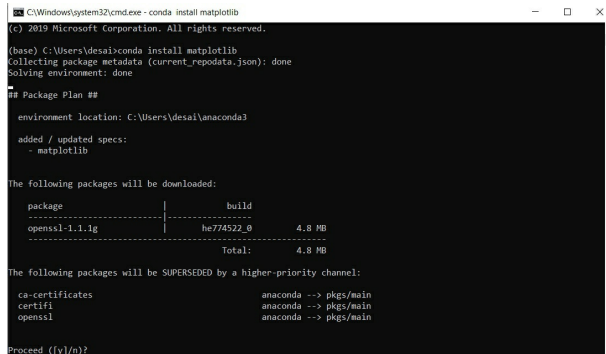
For example, if I were downloading matplotlib, I would type it as the following:



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.18363.1198]
(c) 2019 Microsoft Corporation. All rights reserved.

(base) C:\Users\desai>conda install matplotlib
```

Once you hit enter, it will take you to a screen like this:



```
(c) 2019 Microsoft Corporation. All rights reserved.

(base) C:\Users\desai>conda install matplotlib
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\desai\anaconda3

added / updated specs:
- matplotlib

The following packages will be downloaded:



| package        | build      | size   |
|----------------|------------|--------|
| openssl-1.1.1g | he774522_0 | 4.8 MB |
| Total:         |            | 4.8 MB |



The following packages will be SUPERSEDED by a higher-priority channel:



| package         | channel                |
|-----------------|------------------------|
| ca-certificates | anaconda --> pkgs/main |
| certifi         | anaconda --> pkgs/main |
| openssl         | anaconda --> pkgs/main |



Proceed [y/n]?
```

Type “y” and then hit enter. Repeat this process for each of the libraries

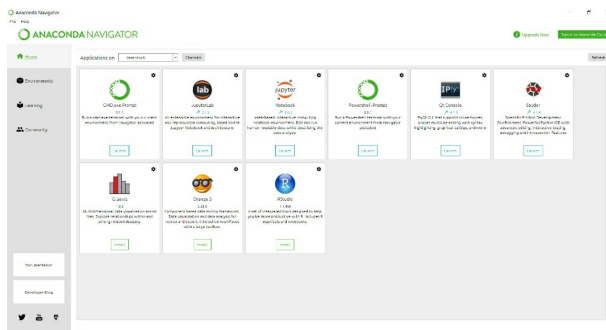
Now we have to get Tensorflow. If you noticed, Tensorflow was not included in the libraries we needed. This is because it has special installation instructions. The commands you need to type in the command line to install Tensorflow are:

First type: “*conda create -n tf tensorflow*” and hit enter. Enter “y” when prompted.

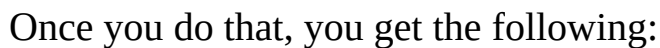
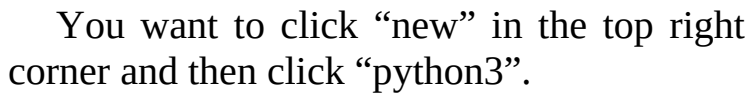
Congrats! You have now downloaded all of the required libraries and software needed for tensorflow and python. If for any reason

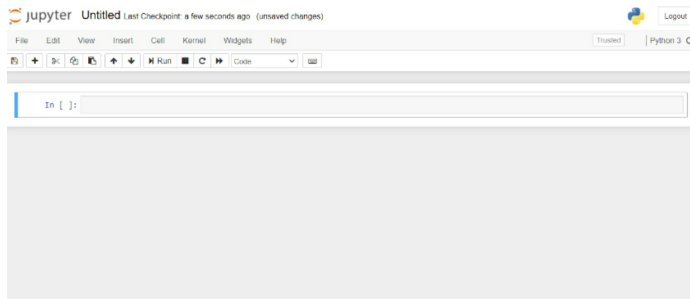
you were unable to download the libraries correctly, look at the pages below which detail how to get google collab setup. As I said before, I recommend Google collab. If you ever end up with a problem that is not related to the code itself, I suggest you switch over to google collab.

The way to open files in anaconda is through the Jupyter notebook. It is the icon in the middle



Once you click launch, a new tab will open in your browser:





You are now done and this is your coding environment

To test if this coding environment runs, type:

```
print("hello world")
```

Then press “shift + enter”. This will run your code and how you will run blocks of code in the future. The words “hello world” should be the output. All the print function does is it outputs whatever you put in parenthesis and the exclamation points.

## Google Collab Setup

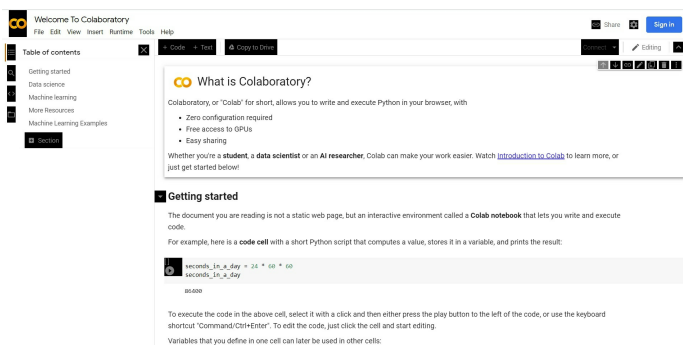


Google collab is the google drive of the coding world. It saves your code in the cloud so that you can access it from virtually anywhere. Collab comes preloaded with all the libraries we will be using so that's great! It also has another major benefit. It comes with an online GPU that you have access to. GPU stands for Graphics Processing unit and what it does is it allows for faster calculations than a CPU (Central processing unit).



Typically, deep learning code is run on a computer's cpu because most people don't have a powerful GPU. However, if you run your code in google collab, that won't matter. You'll have access to the online GPU, so your code will run fast.

The first thing we need to do in the setup process is to go to the following link: <https://colab.research.google.com/notebook>



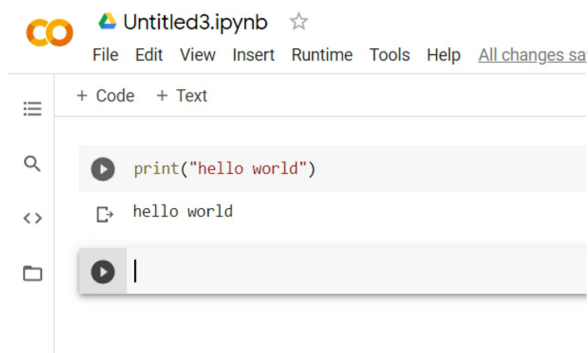
The first thing we need to do is sign in. Make sure you have a google account created and then click the blue button called “Sign in” in the top right.

Once you sign in, you should return back to the previous screen. Congrats! You’re already done.

To make sure that everything is working, run the following command:

```
print("hello world")
```

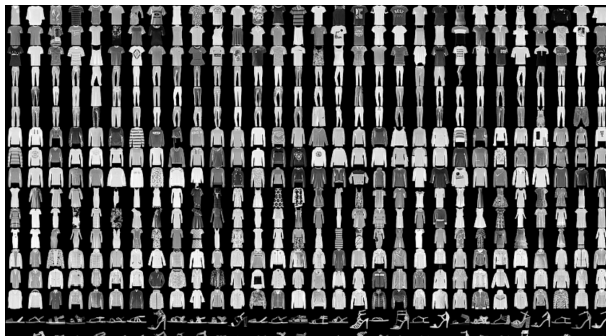
You should get an output like the following:



Your file is automatically saved, but if you want to find it for later make sure to rename your file something you will remember. Congrats, you’ve learned the basics of Google Collab and done the setup! We will now be getting to the fun stuff.

The content from this book has been adapted and modified from tensorflow.org. It is licensed under the [Creative Commons License 4.0](https://creativecommons.org/licenses/by/4.0/). The code has been adapted and modified and is licensed under the [Apache 2.0 License](https://www.apache.org/licenses/LICENSE-2.0).

## Project 1: Image Classification of Clothing



(by Zalando, MIT License)

One of the most basic uses for TensorFlow is image

classification. What Image classification does is it takes a picture as an input and it tries to “guess” what the image is or isn’t. The guess is of course based on an analysis of the image. For example, in this project our program is going to try to classify different items of clothing. However, it is not going to do it randomly. As the project goes on, I will explain what each part of the program does. The code for this project and others can be found in the Google Collab repository(<https://colab.research.google.com/drive/1H1vVHJusp=sharing>) if you would like to look at the completed code. Parts of this code has been adapted from the TensorFlow team.

Begin by opening a new project on Google collab.

The first thing we need to do is import our libraries. The reason we need to import libraries is because our program does not have access to the code from the libraries by default. By importing libraries, we give our program access to this code. We can import using a simple “import” statement followed by the library name. The three libraries we will be using are tensorflow, numpy, and matplotlib. Type the following into Collab:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Make sure not to put any funky spacing as that will mess up Python. To run this code and import the libraries, you can press “Shift + Enter” or press the play button in the top left of the cell. For the next bits of code, you can continue typing

in the current cell with the previous code, or you can click the “+code” button in the top left to create a new cell. This is mainly for organizational purposes.

**Import data:** The next thing we need to do is import our clothes data set and setup our model for train and test data. The dataset we’re going to import contains over 60,000 different images and is called the fashion\_mnist dataset. There are 10 different types of clothing in the dataset and each clothing item is assigned a number (0-9 in order): Tshirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Ankle boot

The model learns with the training data and then checks its results on the test data. We can do this using the following command:

```
fashion_mnist =  
tf.keras.datasets.fashion_mnist  
(train_images, train_labels), (test_images,  
test_labels) = fashion_mnist.load_data()
```

The variable fashion\_mnist holds the data of all the different types of clothing images.

**Data Checking:** We’ve imported the data, but how does it look? Each image is 28 pixels by 28 pixels and we imported 60,000 images into the training set and 10,000 in the test set. We can check our training set size using the following code:

```
train_images.shape
```

As you can see, you get (60000,28,28) which means there’s 60,000 images and each is 28 x 28 in the train set

We can do the same with the test data and type:

```
test_images.shape
```

As you can see, you get (10000,28,28) which means there's 10,000 images and each is 28 x 28 in the test set

**Class names:** Now we need to setup the different types of clothing our model will identify. We can do this with a series of class names. Type the following:

```
class_names = ['Tshirt/top', 'Trouser',  
'Pullover', 'Dress','Coat','Sandal', 'Shirt',  
'Sneaker', 'Bag', 'Ankle boot']
```

**Preprocess Data:** Now that we have imported our data and set it up, we need to preprocess our data before training. The reason we need to preprocess our data is that there are so many images in our dataset. Each image is represented from a value of 0-255. If we kept all of our images in such large numbers, it would take forever to train the model. To fix this, we preprocess the data by dividing all the image data by 255. Type the following:

```
train_images = train_images / 255.0  
test_images = test_images / 255.0
```

We can see how our data has changed by typing the following (don't worry about the complicated code):

```
plt.figure(figsize=(10,10))  
for i in range(25):  
    plt.subplot(5,5,i+1)
```

```
plt.xticks([])
plt.yticks([])
plt.grid(False)
plt.imshow(train_images[i],
cmap=plt.cm.binary)
plt.xlabel(class_names[train_labels[i]])
plt.show()
```



**Building Model:** We will now transition into actually making the model. Type the following code:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,
28)),
    tf.keras.layers.Dense(128,
activation='relu'),
```

```
tf.keras.layers.Dense(10)  
)
```

We are building a sequential model here. What the flatten command does is it takes the 28 X 28 image and makes it one dimensional, so just 784 pixels. The pixels in an image are lined up in one long row instead of being in a 28 X 28 square.

The two dense functions are what generate the layers of the neural network. Each layer is connected. The first dense function creates the first layer and has 128 “neurons” in it. The activation function that is used, relu, is not too important to learn about right now, but what the actual activation function does is. The activation function outputs the values that the model trained into a tangible value.

**Compiling the Model:** Before the model can be trained, a few features must be added to make sure the training works smoothly.

**Loss Function:** The loss function measure how accurate the results of the training were. We want to minimize the output value of the loss function because we want the training to be accurate in guessing correctly.

**Optimizer:** Once the loss function outputs a value, we want to make sure that the model updates to become more accurate based on the training. The optimizer does this for the model.

**Metrics:** Metrics help us track how accurate our model is. We will be using a simple accuracy score.

We will now work on implementing the



steps from the previous page. Type the following code:

```
model.compile(optimizer='adam',loss=tf.keras.losses.SparseCategoricalCrossentropy,metrics=['accuracy'])
```

This is the compile command for your model. The optimizer that was used here was adam. Machine learning topics often have weird wording and this is one of them. Adam is an optimizer that works really well especially when our dataset is very diverse or “noisy”. In the fashion data set, there are many different types of clothing which is why it is considered “noisy”.

The loss function that is used, “SparseCategoricalCrossentropy” works well when we have more than 2 labels. In our case, we are working with 10 different labels which are the different types of clothing.

The last part, just specifies the metric for our model which will be an accuracy score as discussed earlier.

**Training intro:** Although we have setup the basic functionality of the model, it can’t

do anything yet! It hasn't been trained so right now its as dumb as a rock. In this stage, we will work on training the model to detect clothing. To do this, we will be using the training data set from earlier. The steps for training are displayed on the next page.

### **Training steps:**

1. Inputting data into model: The model needs data to train. Luckily, we've already got the data in the form of the *train\_images* and *train\_labels* arrays we made earlier.
2. Making connections: The model learns by fitting the training images to the training labels. For example, it learns that an image that looks like a t-shirt gets a "t-shirt" label.
3. Making predictions: After the model has learned to associate certain labels with certain images, we give it test data and the model tries to predict what the images in the

*test\_images* array are.

4. **Verification:** Once the model has made its “guess” on what the labels of the images in the *test\_images* dataset are, it needs to verify its results so that it can improve itself on the next iteration.

**Training the model:** To train the model, we will be using an in-built function called the “fit” function.

To “call” the function, or in other words use it, we use the following command for our scenario:

```
model.fit(train_images, train_labels, epochs=10)
```

What this does is it runs through step 1-2 for 10 epochs, or 10 times for the entire dataset.

**Accuracy Evaluation:** Now that the model has been trained, we need evaluate how accurate the model was so that we have a sense of how good the model is running. We can do that with the following code:

```
test_loss, test_acc =
```

```
model.evaluate(test_images, test_labels, verbose=2)
print("\nTest accuracy:", test_acc)
```

The *test\_loss* variable measures the loss of the model, while the *test\_acc* measures the accuracy of the model. If you noticed, there is a variable at the end called *verbose*, which is set to a value of 2. All this does is it displays more information about what we trained. We will be using this information in a second.

The output with verbosity should look something like this (it might not look exactly the same:

```
313/313 - 0s - loss: 0.3328 - accuracy: 0.8828
```

```
Test accuracy: 0.8827999830245972
```

If you notice, the test accuracy is slightly less than the accuracy. This occurs because of a phenomenon known as **overfitting**. Overfitting occurs when a model is so trained on a specific dataset, that it is unable to perform as well when it sees a new dataset. In this case, the test data set was totally new and because the model was

overfit on the training data, it performed worse on the test data.

We will get back to overfitting later, but for now we don't have to worry about it too much as the overfitting only slightly affected our model. Remember, the test accuracy was only *slightly* less than the overall accuracy.

**Making predictions:** We can get back to our model now. So far we've built the model, compiled it, trained it, evaluated its accuracy, and now what? Now we get the model to make predictions on the test data. First, we need to normalize the data. We can do that with the following code:

```
probability_model = tf.keras.Sequential([model,  
tf.keras.layers.Softmax()])
```

The softmax function converts the outputs from the model to probabilities. Our model makes predictions about what it thinks each image is in the dataset, however, it is not 100% sure on what an item is, so it estimates the chance that an image is every specific label. For example, it might guess that an

image of a shoe has a 91% of being a shoe. The softmax function converts the linear outputs of the model into a vectorized version so that the model can process the data faster.

The Sequential function takes the outputs of the model and groups in a layer in the probability model.

Now we can work on making predictions for the test images. We will utilize the probability model we built with our previous code. We can do this with the following code:

```
predictions = probability_model.predict(test_images)
```

Now that our predictions for each of the test images has been made, we can take a look at what the model thinks each image is. To view the prediction for the first image, we can use the following code (arrays start at 0, not 1):

```
predictions[0]
```

The output looks like this:

```
array([2.1600704e-06, 1.0047487e-09, 1.3012021e-08,  
4.4011692e-09, 8.2753353e-08, 3.9328373e-04, 9.4819529e-  
07, 6.2794294e-03, 3.7604739e-08, 9.9332410e-01],  
dtype=float32)
```

To see what item the model thinks image 1 is (highest percentage confidence), we can use the following command:

```
np.argmax(predictions[0])
```

We get an output of 9, which means the model thinks image 1 is an ankle boot.

**Verifying Predictions:** Now that the model has made predictions about what it thinks each image is, we can verify if it is correct or not. We will be using a visual representation so that it is easier to understand. Use the following code to setup our visualizer(it's ok if you don't understand it, but I recommend you get this code from

the repository):

```
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(img, cmap=plt.cm.binary)
    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'
    plt.xlabel("{} {} {:.2f}%".format(class_names[predicted_label], 100 * np.max(predictions_array), color))

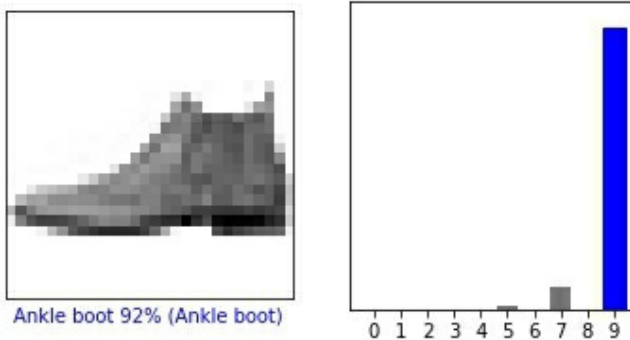
def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array,
color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)
    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
```



Pretty much what these functions do is that they graph the image along with what the model thinks it is confidence wise. If there is a blue bar, it represents a correct prediction. If there is a red bar, it represents an incorrect image.

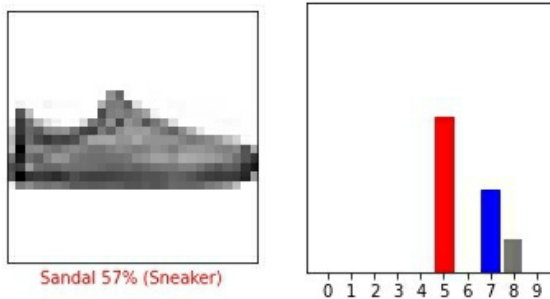
We can now look at what the predictions look like. Using our functions that we defined above, we can do the following to see what the first image looks like:

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```



A correct prediction was made, so the bar is blue. Keep in mind that your accuracy might look slightly different. Let's look at what an incorrect prediction looks like:

```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```



Because the prediction was incorrect, there is a red bar. The blue bar represents the confidence the model had for the correct label. Because it was less than the confidence for the red bar, the red bar was chosen as the label for the image.

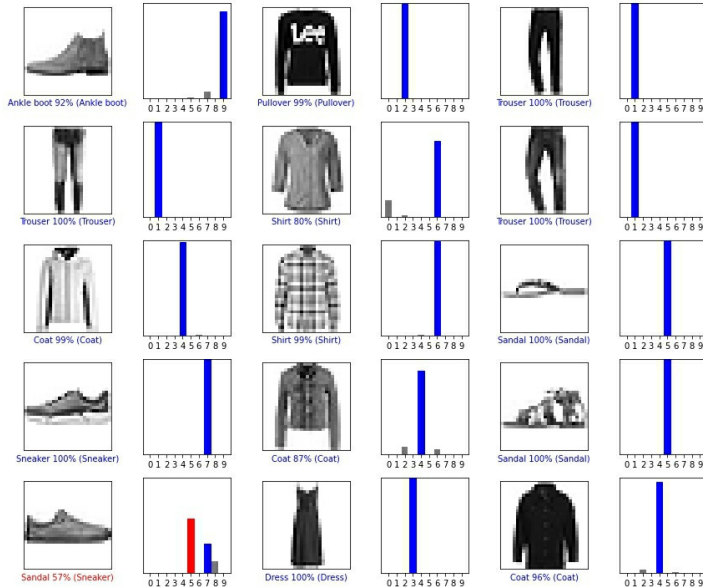
Now that we have seen what a correct and incorrect label looks like, let's plot a couple of images to see what they look like. We can do that with the following code:

```
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
```

```

plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()

```



The images above represent the output of the code. Just as we specified in the code, it is a 3 x 5 image. This visual representation helps us better understand how our model classified each image.

Well, congrats! You just finished your first project: image classification. **You** did this. You created a model that took training data to

improve itself, then you used testing data to see how well the model works and even visually looked at the results. As my father likes to say, this is just the tip of the iceberg. We will be moving on to even more exciting projects. For project 2, we will be shifting focus to a different form of data.

## Project 2: Basic Text Classification



-“It was a **really good** movie!”

The second project we’re going to be working on is Basic Text classification. This comes in the form of movie reviews! We’re going to be taking a dataset of plain text movie reviews and going to be classifying them as good or bad. In essence, we’re doing a sentiment analysis of the reviews. It’s going to be a binary classification (good/bad) this time. It’s different than project 1 which had multiple classifications. Binary classification is used in a variety of machine learning problems such as trying to differentiate a spam email from a real one. We’re going to be looking at the words and train a classifier that is able to identify how the review leans. The set that we’re going to be using comes from IMDb. Like project 1, the code for project 2 can also be found in the Google Collab repository.

<https://colab.research.google.com/drive/1M7iC69qe2Si1dzp3kqODlNMRv?usp=sharing>

**Importing Libraries:** The first thing we're going to be doing is importing the libraries that we need. We can do that with the following code:

```
import matplotlib.pyplot as plt
import os
import re
import shutil
import string
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import losses
from tensorflow.keras import preprocessing
from tensorflow.keras.layers.experimental.preprocessing
import TextVectorization
```

**Importing Data:** The next thing we need to do is import our data that we will be training on. The dataset contains a total of 50,000 movie reviews, but we will be splitting it up into 2 parts for training and testing. 25,000 of the reviews will be for training and 25,000 of the reviews will be for testing. When the training set is equal to the testing set, the dataset is known as being balanced. Unlike the dataset from project 1,

the dataset we will be using here is not built into the TensorFlow library. Instead, we will be downloading it from online from the Stanford AI website. The code is on the next page.

```
url =  
"https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.ta  
dataset = tf.keras.utils.get_file("aclImdb_v1", url,  
untar=True, cache_dir='.',cache_subdir="")  
dataset_dir = os.path.join(os.path.dirname(dataset),  
'aclImdb')
```

The file for the data is named “aclImdb\_v1”. Another operation that we’re doing here is doing some file management. The `os.path.join` concatenates the `aclImdb_v1` file to the end of the dataset path.

Now, we will be messing around with some of the files so we get more familiar with what is in the dataset and how we can use it.



First off, we can see what's in the directory with the following command:

```
os.listdir(dataset_dir)
```

As you can see, there are 5 files(['train', 'imdbEr.txt', 'imdb.vocab', 'README', 'test']) in the directory.

The following command creates a path to training directory and the second command lists the files in the directory.

```
train_dir = os.path.join(dataset_dir, 'train')
```

```
os.listdir(train_dir)
```

The output looks like this:

```
['urls_neg.txt',  
'pos',  
'unsup',  
'unsupBow.feats',  
'labeledBow.feats',  
'urls_unsup.txt',  
'urls_pos.txt',  
'neg']
```

The ‘pos’ and ‘neg’ directories have positive and negative reviews in the form of text files respectively.

Let’s take a quick look at what one of these text files looks like. We can do this with the following commands:

```
sample_file = os.path.join(train_dir, 'pos/412_8.txt')
with open(sample_file) as f:
    print(f.read())
```

After running the command, you should see something like this:

```
I liked this movie because it told a very interesting story
about living in a totally different world at the south pole.
Susan Sarandon is such a good actor, that she made an
interesting, strong character out of mediocre writing
```

**Loading the Data:** Now that we have imported the data, we need to prepare it so that we can use it to train and test. We will do this using the `text_dataset_from_directory` command which will turn our text data from the pos and neg directories into datasets. The `text_dataset_from_directory` needs a directory structure in the following format:

```
main_directory/
```

```
...class_a/  
.....a_text_1.txt  
.....a_text_2.txt  
...class_b/  
.....b_text_1.txt  
.....b_text_2.txt
```

We have our two directories which are the pos and neg directories that we created earlier. However, there are additional directories in the main directory in which the pos and neg directories are located. We need to remove these so we are only left with the pos and neg directories.

We can do this with the following command:

```
remove_dir = os.path.join(train_dir, 'unsup')  
shutil.rmtree(remove_dir)
```

Now, we will once again use the `text_dataset_from_directory` command to create a `tf.data.Dataset`. This type of dataset makes it easy for us to apply operations over all of our data if we want to transform our data.

When training machine learning models, we split our data into three types: training, testing, and validation data. We have already gone over training and testing data, but validation data is something new. When a model is training, validation data acts as a “wrench” in the system by providing the learning model new data to train on that it hasn’t seen before. The main reason this is done is to reduce overfitting of a model to the training data.

The dataset that we are using (The IMDB dataset) is already split into training and testing data; however, it is missing the third category which is validation data. We will create the validation data by splitting the training data in 80% training data and 20% validation data. We will first start by creating the training data split.

We can do this with the following commands:

```
batch_size = 32
seed = 42

raw_train_ds =
```

```
tf.keras.preprocessing.text_dataset_from_directory(  
    'aclImdb/train',  
    batch_size=batch_size,  
    validation_split=0.2,  
    subset='training',  
    seed=seed)
```

The batch size is the number of text examples that are passed into the model. The seed makes sure the model starts at the same point each time it trains.

The output will look like this:

```
Found 25000 files belonging to 2 classes.  
Using 20000 files for training.
```

As mentioned above, 80% or 20,000 of the files will be used for training while the 20% will be used for validation.

To take a look at a few of the examples, you can use the following commands:

```
for text_batch, label_batch in raw_train_ds.take(1):  
    for i in range(3):  
        print("Review", text_batch.numpy()[i])  
        print("Label", label_batch.numpy()[i])
```

If you notice from the output, there are random characters and “0’s” and “1’s” associated with each training example.

Those labels correspond to pos and neg reviews. To which is which, you can use the following commands:

```
print("Label 0 corresponds  
to",raw_train_ds.class_names[0])  
print("Label 1 corresponds to",raw_train_ds.class_names[1])
```

The output looks like this:

```
Label 0 corresponds to neg  
Label 1 corresponds to pos
```

So, in the future, you know that a 0 label corresponds to neg and a 1 label corresponds to pos.

Just like we created the training data split, we will create a split for the validation data and testing data. As seen before, we used 80% or 20,000 examples for the training split out of the 25,000 total in directory. We will use the remaining 20% or 5,000 examples for the validation data set.

We can do that with the following commands:

```
raw_val_ds =  
tf.keras.preprocessing.text_dataset_from_directory(  
    'aclImdb/train',  
    batch_size=batch_size,  
    validation_split=0.2,  
    subset='validation',  
    seed=seed)
```

The output looks like this:

```
Found 25000 files belonging to 2 classes.  
Using 5000 files for validation.
```

As mentioned before, the 5000 files are allocated to validation.

Now we will be doing the same for the test data, but this time we won't be splitting the data as the test data is allocated only for test data.

```
raw_test_ds =  
tf.keras.preprocessing.text_dataset_from_directory(  
    'aclImdb/test',  
    batch_size=batch_size)
```

The output looks like:

Found 25000 files belonging to 2 classes.

If you noticed, the difference between this code and the code before is that the file path is 'aclImdb/test' vs 'aclImdb/train'.

### **Preparing the Dataset for Training:**

Now we will be doing three things to the data: standardization, tokenization, and vectorization. These may seem like obscure words, but they mean really simple things.

Standardization is the process of preprocessing our text. If you noticed from earlier, when we printed a few of the examples out, there were a few extra characters in the text like `<br />`. This is because the text was taken online from IMDB where the text is formatted with HTML which is a different language.

There is a default standardizer in the TextVectorization layer, but it doesn't catch the extra html code so we will be using a custom standardizer to preprocess our data.



```
def custom_standardization(input_data):  
    lowercase = tf.strings.lower(input_data)  
    stripped_html = tf.strings.regex_replace(lowercase, '<br />',  
    ')  
    return tf.strings.regex_replace(stripped_html,  
                                    ' [%s]' % re.escape(string.punctuation),  
                                    ")
```

This standardizer removes all the unnecessary characters and makes everything lowercase.

The other two operations that we will be doing are tokenization and vectorization. Tokenization splits strings into tokens. For example, splitting a sentence (a string) into individual words (tokens) would be tokenization. Vectorization takes the tokens and converts tokens into numerical values so they can be fed into the model.

We will now be creating a TextVectorization layer that we will use to standardize, tokenize, and vectorize our data all at once. We will set the the output\_mode to int so that we get a unique integer for each and every token.

We will be using our custom

standardization operation that we defined before. We will also be setting some constant parameters. We set the `max_features` to 10,000 and the sequence length to 250. The `max_features` limits the number of features that the model can consider and the sequence length restricts the total length of a sequence that is passed into the model for training.

```
max_features = 10000
sequence_length = 250

vectorize_layer = layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=max_features,
    output_mode='int',
    output_sequence_length=sequence_length)
```

Next, we will call “adapt” which fits a preprocessing layer to a dataset. This makes it so that model makes an index for each string to an integer.

This can be done with the following code:

```
train_text = raw_train_ds.map(lambda x, y: x)
vectorize_layer.adapt(train_text)
```

Now that we have “adapted” the data, lets see the results by creating a function and using it. We can create the function using the following code:

```
def vectorize_text(text, label):  
    text = tf.expand_dims(text, -1)  
    return vectorize_layer(text), label
```

We can now visualize a few of the examples using the following code:

```
text_batch, label_batch = next(iter(raw_train_ds))  
first_review, first_label = text_batch[0], label_batch[0]  
print("Review", first_review)  
print("Label", raw_train_ds.class_names[first_label])  
print("Vectorized review", vectorize_text(first_review,  
first_label))
```

The long output shows that each of the tokens (strings) has been replaced by a corresponding integer. You can see what integer represents by calling the `get_vocabulary()` command. You can do that with the following code:

```
print("1287 ---> ",vectorize_layer.get_vocabulary()  
[1287])  
print(" 313 ---> ",vectorize_layer.get_vocabulary()[313])
```

```
print('Vocabulary size:
{}'.format(len(vectorize_layer.get_vocabulary())))
```

The output looks like this:

```
1287 ---> silent
313 ---> night
Vocabulary size: 10000
```

As a final step before training our model, we will apply the textvectorization layer we created before to our data. We can do that with the following code:

```
train_ds = raw_train_ds.map(vectorize_text)
val_ds = raw_val_ds.map(vectorize_text)
test_ds = raw_test_ds.map(vectorize_text)
```

**Configuring the Dataset to Increase Performance:** When dealing with such large amounts of data, performance issues arise. To minimize bottlenecks, we need to do a few things first.

The `.cache()` function will keep our data in memory after it is loaded. This reduces the bottleneck of the dataset loading. We won't

go too much into how memory is stored as that's not the main focus of our project, but just know that this function increases efficiency when related to a dataset loading.

The `.prefetch()` function overlaps preprocessing data and model execution so that training is done faster.

We can apply the two functions mentioned earlier on our data using the following code:

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds =
train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

## **Creating the Model...**

It is finally time to create our model after all the steps we conducted earlier!

We can do that with the following code:

```
embedding_dim = 16
```

```
model = tf.keras.Sequential([
layers.Embedding(max_features + 1, embedding_dim),
layers.Dropout(0.2),
layers.GlobalAveragePooling1D(),
layers.Dropout(0.2),
layers.Dense(1)])
model.summary()
```

The model goes through a sequence of steps. That are explained on the next page.

### Sequential steps:

1. **Embedding Layer:** This layer takes the integers that we assigned to each token and looks up an embedding vector for each word. These vectors are learned when the model trains. The vectors add extra dimensions to the output array. The dimensions of the vector are (batch, sequence, embedding)
2. **Dropout Layer:** The dropout layer

randomly sets the input values to 0. This prevents overfitting by “throwing a wrench” into the model so it’s not too adapted to the training examples. In this case, it’s set to 0.2.

3. Global Average Pooling 1D: This returns an output vector of the same size for each example by averaging across all the examples. This makes it that although the training examples might be varying in length, the model is able to handle all of them the same.
4. Dense layer: The output vector is then put through a dense layer. The dense layer is connected to all the neurons of the layers before it, which is why it’s called the dense layer. It makes it so that each layer receives input from the previous layer. The dense layer has a single output node.

## **Loss Function and Optimization:**

The model needs a loss function and optimizer to improve itself. After training on examples, the model needs a way to improve itself from what it has learned. This is done with the loss function and optimizer.

Our problem is a binary classification problem. This means that when classifying our reviews, the model outputs probabilities for each review on whether it is good or bad. Because our model is binary classification and it outputs probabilities, we will be using the `BinaryCrossentropy` loss function. The optimizer we will be using is `adam`. Adam will help change the model to improve it based on what the loss function outputs.



We can use the loss function and the optimizer with the following code:

```
model.compile(loss=losses.BinaryCrossentropy(from_logits=True),  
              optimizer='adam',  
              metrics=tf.metrics.BinaryAccuracy(threshold=0.0))
```

**Training the Model:** Now that we have setup the model, we can begin training it. We can train it with the following code:

```
epochs = 10  
history = model.fit(
```

```
train_ds,  
validation_data=val_ds,  
epochs=epochs)
```

We set the epochs to 10, so the model will train for 10 iterations. We inputted our files that we created before. train\_ds is the training data while val\_ds is the validation data.

The output will look something like this:

```
Epoch 1/10  
625/625 [=====] - 3s  
4ms/step - loss: 0.6633 - binary_accuracy: 0.6982 - val_loss:  
0.6142 - val_binary_accuracy: 0.7726  
Epoch 2/10  
625/625 [=====] - 2s  
3ms/step - loss: 0.5479 - binary_accuracy: 0.8020 - val_loss:  
0.4978 - val_binary_accuracy: 0.8230  
  
• • •  
Epoch 9/10  
625/625 [=====] - 2s  
3ms/step - loss: 0.2457 - binary_accuracy: 0.9103 - val_loss:  
0.2961 - val_binary_accuracy: 0.8778  
Epoch 10/10  
625/625 [=====] - 2s  
3ms/step - loss: 0.2318 - binary_accuracy: 0.9166 - val_loss:  
0.2915 - val_binary_accuracy: 0.8784
```

(A few of the epochs in-between have been skipped to save space)

**Evaluating the Model:** Now that we have trained the model, let's see how well it performs. We will have two values when evaluating the model. There is the loss which is a number that represents our error. Lower numbers are better as that means less of an error. Then there is the accuracy which is how well the model performed in classifying the movie reviews.

We can evaluate our model with the following code:

```
loss, accuracy = model.evaluate(test_ds)

print("Loss: ", loss)
print("Accuracy: ", accuracy)
```

The output will look similar to this:

```
Loss: 0.30995428562164307
```

```
Accuracy: 0.8739200234413147
```

With just a little bit of training, we were able to achieve an 87.39% accuracy when our model was trying to identify positive and

negative reviews.

If you increase the number of epochs, the model will achieve a higher accuracy; however, it will take more time to train. That's one of the tradeoffs when it comes to even bigger models. It sometimes takes days or weeks to train a model because of how large they are and how much data a scientist or engineer is working with. They have to optimize time it takes to train vs efficiency of the model.

**Plot of Accuracy Vs Time:** We will now be creating a graph to show how the accuracy of the model changed over time. This helps visualize what is happening as the model is running.

In order to graph the data as the model was running, we need a way to access the data. This data is stored in the History object that we created earlier when doing `model.fit()`.

Enter the following code to setup the history object:

```
history_dict = history.history
history_dict.keys()
```

The output should look like this:

```
dict_keys(['loss', 'binary_accuracy', 'val_loss',
'val_binary_accuracy'])
```

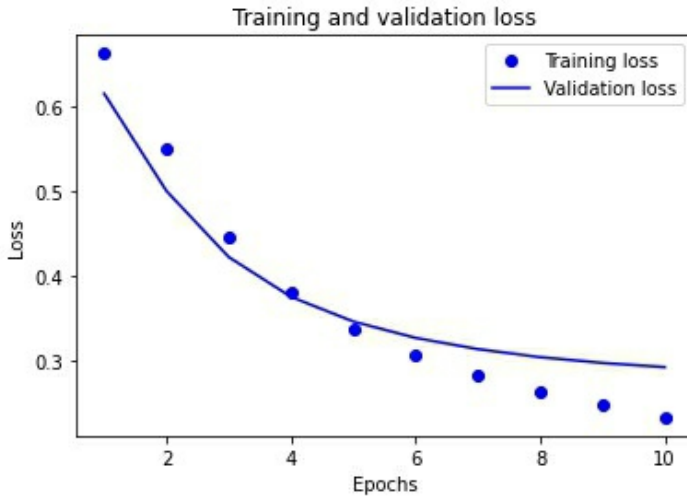
If you notice, there are a total of four metrics that were monitored during training.

We can plot Loss with the following code:

```
acc = history_dict['binary_accuracy']
val_acc = history_dict['val_binary_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title("Training and validation loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

The output looks like this:



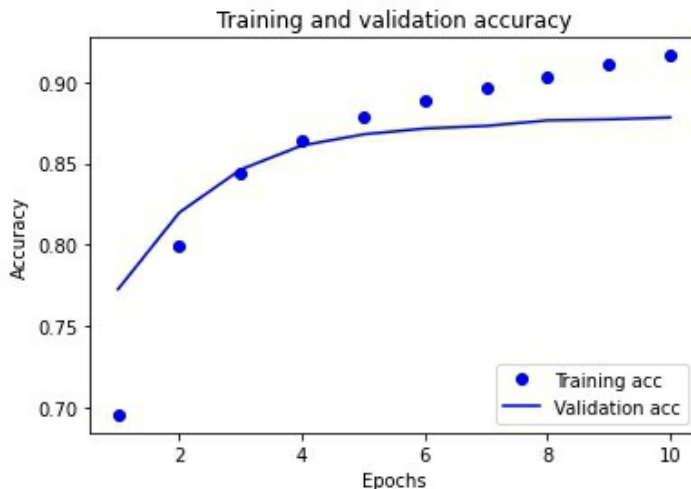
As you can see, the training loss decreased as the model trained more. This means that our model improved as time went and each epoch was over.

Now, we need to visualize the accuracy over time. We can do that with the following code:

```
plt.plot(epochs, acc, 'bo', label='Training acc')
```

```
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```

The output looks like this:



The training accuracy increased after each epoch meaning that our model was getting better. We want training accuracy to increase and training loss to decrease. After each epoch, the change in the values decreases. For example, at the beginning, the training

accuracy greatly increased, but then it leveled off after every successive epoch. The same thing happened for the training loss. It decreased drastically in the beginning, but then decreased at a slower rate as time went on.

The validation loss and accuracy had a slightly different trend than the training loss and accuracy. The validation loss hit its bottom while the training loss kept decreasing. And, the validation accuracy reached its peak while the training accuracy was still increasing. This means that the model is overfitting because it performs better on the training data than the validation data (new data),

One way to stop overfitting in this case is to stop training when the validation accuracy peaks. This prevents the model from overtraining and continually overfitting the training data. An easy way to do this is by implementing the `tf.keras.callbacks.EarlyStopping` callback,



however, we won't be getting into that.

**Exporting the Model:** Our model is done! Instead of just wasting it, we can save it and use it on future data. We will now work on saving and exporting our model.

We need to do some modifications to our model before we export it. While we were building our model, we used a TextVectorization layer on our data before we passed it into our model. But, if we export our model how it is now and use it, the data won't have gone through the TextVectorization layer. To fix this, we need to put the TextVectorization layer in our model. To do this we will be creating a new model that is identical to our old model except it includes the TextVectorization layer.

We can do that with the following code:

```
export_model = tf.keras.Sequential([
    vectorize_layer,
    model,
    layers.Activation('sigmoid')
```

```

])

    export_model.compile(
        loss=losses.BinaryCrossentropy(from_logits=False),
        optimizer="adam", metrics=['accuracy']
    )

    loss, accuracy = export_model.evaluate(raw_test_ds)
    print(accuracy)

```

Now that we have exported our new and improved model, we can actually use it!

All we have to do to use our model is type:  
`export_model.fit()`

We need some examples to use our model on, so let's make some.

We can do so with the following code:

```

examples = [
    "The movie was awesome",
    "The movie was good",
    "The movie was terrible..."
]

```

Now all we have to do is type the

following:

```
export_model.predict(examples)
```

Our output should look something like this:

```
array([[0.58980715],  
       [0.54284245],  
       [0.3705216]], dtype=float32)
```

Congratulations once again! Project 2 has been completed. You took a dataset (IMDB movie reviews) off the internet and downloaded it into your program.

You then built and trained a model that identifies positive and negative reviews through sentiment analysis.

We will now continue on to project 3.

## Project 3: Text Classification with TensorFlow Hub



-“It was a **really good** movie!”

### Or was it?

Project 2 dealt with how to do basic text classification, and it involved many steps. With project 3, we’re going to do a different slightly more advanced version that requires less steps and works better. We will do this by using TensorFlow Hub.

[TensorFlow Hub](#) is a repository of machine learning models. It’s incredibly useful for our cause because we will be using a machine learning model meant for classifying text. There’s tons of other resources on the website, so check it out!

We will once again be doing sentiment analysis through binary classification, so let’s get started! The code for this project can be

found at:

[https://colab.research.google.com/drive/1i\\_tJiSZXRKyvE\\_?usp=sharing](https://colab.research.google.com/drive/1i_tJiSZXRKyvE_?usp=sharing)

Let's first start by importing the required libraries:

```
import os
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
```

As you can see, we are importing TensorFlow hub into our code among other things to use.

### **importing our IMDB data:**

```
train_data, validation_data, test_data = tfds.load(
    name="imdb_reviews",
    split=('train[:60%]', 'train[60%:]', 'test'),
    as_supervised=True)
```

As you can see, there is a split in our data. The training data has been split into a 60/40

split for training and validation respectively.

Let's try to understand some of the data that we've imported. We can look at the first 10 examples with the following code:

```
train_examples_batch, train_labels_batch =  
next(iter(train_data.batch(10)))  
train_examples_batch
```

The output should look something like this (only the first few examples have been included to save space):

```
<tf.Tensor: shape=(10,), dtype=string, numpy=  
array([b"This was an absolutely terrible movie. Don't be  
lured in by Christopher Walken or Michael Ironside. Both  
are great actors, but this must simply be their worst role in  
history. Even their great acting could not redeem this movie's  
ridiculous storyline. This movie is an early nineties US  
propaganda piece. The most pathetic scenes were those when  
the Columbian rebels were making their cases for  
revolutions. Maria Conchita Alonso appeared phony, and her  
pseudo-love affair with Walken was nothing but a pathetic  
emotional plug in a movie that was devoid of any real  
meaning. I am disappointed that there are movies like this,  
ruining actor's like Christopher Walken's good name. I could  
barely sit through it."],
```

b'I have been known to fall asleep during films, but this is usually due to a combination of things including, really tired, being warm and comfortable on the sette and having just eaten a lot. However on this occasion I fell asleep because the film was rubbish. The plot development was constant. Constantly slow and boring. Things seemed to happen, but with no explanation of what was causing them or why. I admit, I may have missed part of the film, but i watched the majority of it and everything just seemed to happen of its own accord without any real concern for anything else. I cant recommend this film at all.',

b'Mann photographs the Alberta Rocky Mountains in a superb fashion, and Jimmy Stewart and Walter Brennan give enjoyable performances as they always seem to do. <br /><br />But come on Hollywood - a Mountie telling the people of Dawson City, Yukon to elect themselves a marshal (yes a marshal!) and to enforce the law themselves, then gunfighters battling it out on the streets for control of the town? <br /><br />Nothing even remotely resembling that happened on the Canadian side of the border during the Klondike gold rush. Mr. Mann and company appear to have mistaken Dawson City for Deadwood, the Canadian North for the American Wild West.<br /><br />Canadian viewers be prepared for a Reefer Madness type of enjoyable howl with this ludicrous plot, or, to shake your head in disgust.',

As you can see, the data is raw an unedited and still contains all of extra characters, however, we will be dealing with that later. For now, we're focused on how the data is

formatted and how it is labeled.

In fact, we can take a look at what the first 10 labels are with the following code:

```
train_labels_batch
```

The output looks like this:

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([0, 0, 0, 1, 1, 1, 0, 0, 0, 0])>
```

There are 0's and 1's attached to each movie review. 0 Means a negative movie review and 1 means a positive movie review.

**Building the Model:** We will now work on building our model which will be easier than last time. We have a couple things to consider when constructing our model:

1. How to represent data
2. How many layers will we use in the model



3. How large do we want to make our model

Because our data is all text, one way we can represent our data to the model is as embedding vectors. An embedding vector takes words from our sentences and represents similar words together. This method provides us with several benefits:

1. We don't have to worry about text preprocessing as embedding eliminates the need for preprocessing.
2. Benefit from previous training. The embedding layer has already been trained previously because we're taking it from TensorFlow.hub. This saves us time as we don't have to do it ourselves.
3. The embedding layer makes it so that all the data is the same size so

that the model has an easier time training on large amounts of data.

For this project, we will be using a **pre-trained text embedding model** from TensorFlow Hub called [google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2).

There are several other text embedding models from TensorFlow Hub, but we will be sticking with this one.

Let's create the embedding layer. We can do that with the following code:

```
embedding = "https://tfhub.dev/google/nnlm-en-  
dim50/2"  
hub_layer = hub.KerasLayer(embedding, input_shape=  
[],  
                             dtype=tf.string, trainable=True)  
hub_layer(train_examples_batch[:3])
```

The output should look like this (cut short to save space):

```
<tf.Tensor: shape=(3, 50), dtype=float32, numpy=  
array([[ 0.5423195 , -0.0119017 ,  0.06337538,
```

```
0.06862972, -0.16776837,  
        -0.10581174, 0.16865303, -0.04998824,  
-0.31148055, 0.07910346,  
        0.15442263, 0.01488662, 0.03930153, 0.1977271  
-0.12215476,  
        -0.04120981, -0.2704109 ,  
-0.21922152, 0.26517662, -0.80739075,  
        0.25833532, -0.3100421 , 0.28683215, 0.1943387  
, -0.29036492,  
        0.03862849, -0.7844411 , -0.0479324 , 0.4110299  
, -0.36388892,  
        -0.58034706, 0.30269456, 0.3630897 ,  
-0.15227164, -0.44391504,  
        0.19462997, 0.19528408, 0.05666234, 0.2890704  
, -0.28468323,  
        -0.00531206, 0.0571938 , -0.3201318 ,  
-0.04418665, -0.08550783,
```

Let's now build the full model with the following code:

```
model = tf.keras.Sequential()  
model.add(hub_layer)  
model.add(tf.keras.layers.Dense(16, activation='relu'))  
model.add(tf.keras.layers.Dense(1))  
  
model.summary()
```

The output looks like this:

Model: "sequential"		
Layer (type)	Output Shape	Param #
=====		
keras_layer (KerasLayer)	(None,	
50) 48190600		
dense (Dense)	(None, 16)	816
dense_1 (Dense)	(None, 1)	17
=====		
Total params: 48,191,433		
Trainable params: 48,191,433		
Non-trainable params: 0		
=====		

In the code, the first layer is a pretrained saved model from TensorFlow Hub that is used to map a string into an embedding vector. The pretrained model that we are using splits each sentence into tokens and then embeds each token and then combines them all together. The vector we get is then sized

Number of examples X Embedding Dimension.

The output vector generated from the

embedding layer is a fixed vector that is then put through the densely connected layer of 16 hidden units or nodes.

The last layer is densely connected also and outputs through a single node.

**Loss Function and Optimizer:** A model is built, but it still needs its loss function and optimizer to improve over time. Because this is a binary classification problem, we will be using the `binary_crossentropy` loss function for our model.

There are several loss functions that could be used, but the one that'll probably work the best for binary classification is `binary_crossentropy`.

We can make our loss function with the following code:

```
model.compile(optimizer='adam',  
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),  
              metrics=['accuracy'])
```

**Training the Model:** We will be training

our model for 10 epochs or iterations with a batch size of 512 meaning that 512 examples will be passed in the model for every epoch.

We can train the model with the following code:

```
history = model.fit(train_data.shuffle(10000).batch(512),  
                    epochs=10,  
                    validation_data=validation_data.batch(512),  
                    verbose=1)
```

The output will look something like this:

```
Epoch 1/10  
30/30 [=====] - 4s  
69ms/step - loss: 0.6458 - accuracy: 0.5411 - val_loss:  
0.5995 - val_accuracy: 0.5900  
Epoch 2/10  
30/30 [=====] - 4s  
100ms/step - loss: 0.5359 - accuracy: 0.6899 - val_loss:  
0.4956 - val_accuracy: 0.7347  
Epoch 3/10  
30/30 [=====] - 3s  
66ms/step - loss: 0.4049 - accuracy: 0.8167 - val_loss:
```

0.3975 - val\_accuracy: 0.8207  
Epoch 4/10  
30/30 [=====] - 3s  
70ms/step - loss: 0.2929 - accuracy: 0.8873 - val\_loss:  
0.3465 - val\_accuracy: 0.8527  
Epoch 5/10  
30/30 [=====] - 3s  
66ms/step - loss: 0.2158 - accuracy: 0.9233 - val\_loss:  
0.3199 - val\_accuracy: 0.8576  
Epoch 6/10  
30/30 [=====] - 3s  
67ms/step - loss: 0.1602 - accuracy: 0.9487 - val\_loss:  
0.3097 - val\_accuracy: 0.8625  
Epoch 7/10  
30/30 [=====] - 3s  
67ms/step - loss: 0.1176 - accuracy: 0.9656 - val\_loss:  
0.3083 - val\_accuracy: 0.8681  
Epoch 8/10  
30/30 [=====] - 3s  
64ms/step - loss: 0.0864 - accuracy: 0.9779 - val\_loss:  
0.3140 - val\_accuracy: 0.8671  
Epoch 9/10  
30/30 [=====] - 3s  
68ms/step - loss: 0.0627 - accuracy: 0.9875 - val\_loss:  
0.3216 - val\_accuracy: 0.8692  
Epoch 10/10  
30/30 [=====] - 3s  
66ms/step - loss: 0.0457 - accuracy: 0.9922 - val\_loss:  
0.3329 - val\_accuracy: 0.8688

**Evaluating the Model:** We can now see how the model performs. We can see that based on the two values that we receive: accuracy and loss.

To evaluate the model, we can type the following code:

```
results = model.evaluate(test_data.batch(512),  
verbose=2)  
  
for name, value in zip(model.metrics_names, results):  
    print("%s: %.3f" % (name, value))
```

The output looks like this:

```
49/49 - 2s - loss: 0.3597 - accuracy: 0.8541  
loss: 0.360  
accuracy: 0.854
```

With just a little training, the model received an accuracy of over 85%. This was done of course with very little training. We only did 10 iterations. If the model was trained even more, it would reach up to 95% accuracy.



Project 3 is now complete, and it was much simpler than project 2. We used preexisting code to speed up our project. In the developer world, lots of code is reused through things such as libraries, repositories, etc. It's all about reusing, reducing, and recycling.

## Project 4: Predicting fuel efficiency



The aim of regression is to predict the value of an input based on previous data. In our case, we will be predicting the fuel efficiency of cars. We will be using the Auto MPG dataset to train our model. The dataset contains data from cars from the 1970's and

1980's. We will provide our model with many different features of a car such as number of cylinders, horsepower, weight, etc. and the model will be responsible for outputting the fuel efficiency (MPG) of the car. Let's get started on the next page. The code for this project can be found at: <https://colab.research.google.com/drive/1IIZns8?usp=sharing>

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
np.set_printoptions(precision=3, suppress=True)
```

We will first begin by importing our libraries. If you notice, there are new libraries this time. That is the Seaborn library and the pandas library. The seaborn library is a data visualization library that works in conjunction with matplotlib. The Pandas library helps frame and manage our data better.

The last command `np.set_print_options` makes outputs from numpy easier to read.

Next, we will be importing tensorflow:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

**Downloading Data:** The first thing we need to do is download our data. We will be downloading our Auto MPG dataset from the UC Irvine website. We can do that with the following code:

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-  
databases/auto-mpg/auto-mpg.data'  
column_names = ['MPG', 'Cylinders', 'Displacement',  
'Horsepower', 'Weight',  
                'Acceleration', 'Model Year', 'Origin']
```

```
raw_dataset = pd.read_csv(url, names=column_names,  
                           na_values='?', comment='\t',  
                           sep=' ', skipinitialspace=True)
```

We have set up our column names with the information we will be entering into our model to train it.

The next thing we need to do is create our data set. We can do that with the following code:

```
dataset = raw_dataset.copy()  
dataset.tail()
```

## **Cleaning Data:**

The dataset contains some unknown values, so we can clean that with the following code:

```
dataset.isna().sum()
```

The output looks like this:

```
MPG      0
Cylinders 0
Displacement 0
Horsepower 6
Weight    0
Acceleration 0
Model Year 0
Origin    0
dtype: int64
```

To keep things simple, we're gonna drop a row, so you can do this with the following code:

```
dataset = dataset.dropna()
```

The origin category is not a numerical value, so we need to encode values to each of the countries of origin that are in the dataset. We can do that with the following code:

```
dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3: 'Japan'})
```

As you can see, each country is given a unique number. For example, USA is 1.

We can quickly see the result of this through the following code:

```
dataset = pd.get_dummies(dataset, columns=['Origin'],
prefix="", prefix_sep="")
dataset.tail()
```

The output looks like this (single table has been split into two because it doesn't fit on one page):

MPG	Cylinders	Displacement	Horsepower
27.0	4	140.0	86
44.0	4	97.0	52
32.0	4	135.0	84
28.0	4	120.0	79
31.0	4	119.0	81

Europe	Japan	USA
0	0	1
1	0	0

## Split Data Into Training and Test Sets:

We will now be splitting our data into the two categories. Training will be used to train, and test set will be used at the end.

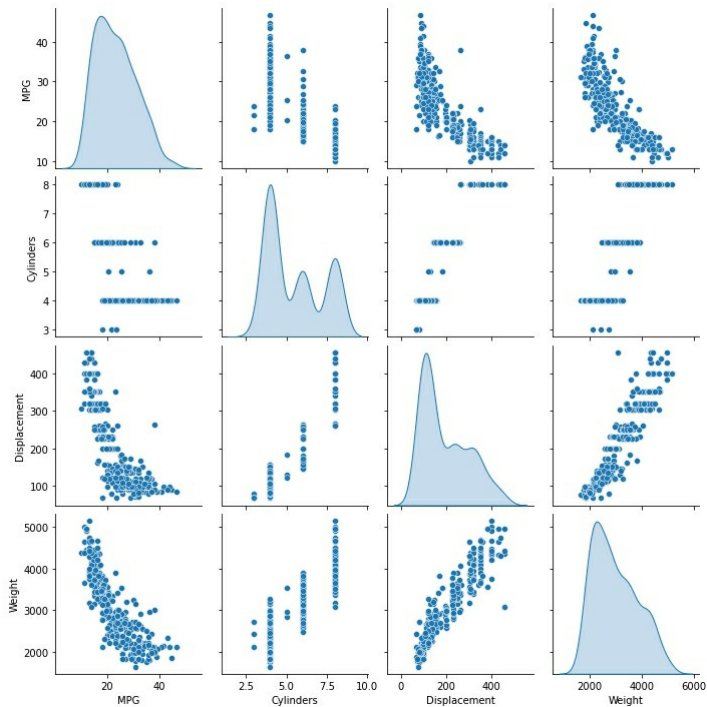
We can do that with the following code:

```
train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

### **Inspect the Data:**

We can see what our data looks like with the following code:

```
sns.pairplot(train_dataset[['MPG', 'Cylinders',  
'Displacement', 'Weight']], diag_kind='kde')
```



As seen, all the data is visualized on different graphs

We can also check overall statistics with the following code:



```
train_dataset.describe().transpose()
```

The output looks like this:

MPG	314.0	23.310510	7.728652	10.0	17.00	22.0	28.95	46.6
Cylinders	314.0	5.477707	1.699788	3.0	4.00	4.0	8.00	8.0
Displacement	314.0	195.318471	104.331589	68.0	105.50	151.0	265.75	455.0
Horsepower	314.0	104.869427	38.096214	46.0	76.25	94.5	128.00	225.0
Weight	314.0	2990.251592	843.898596	1649.0	2256.50	2822.5	3608.00	5140.0
Acceleration	314.0	15.559236	2.789230	8.0	13.80	15.5	17.20	24.8
Model Year	314.0	75.898089	3.675642	70.0	73.00	76.0	79.00	82.0
Europe	314.0	0.178344	0.383413	0.0	0.00	0.0	0.00	1.0
Japan	314.0	0.197452	0.398712	0.0	0.00	0.0	0.00	1.0
USA	314.0	0.624204	0.485101	0.0	0.00	1.0	1.00	1.0

## Splitting Features From Labels:

We are separating the label from the features as the label is the value that the model will train with. We can do that with the following code:

```
train_features = train_dataset.copy()
test_features = test_dataset.copy()

train_labels = train_features.pop('MPG')
test_labels = test_features.pop('MPG')
```

## Normalization:

Before we normalize our data, let's take a look at how varying our data is in terms of ranges. We can do that with the following code:

```
train_dataset.describe().transpose()[['mean', 'std']]
```

The output looks like this:

		mean	std
MPG	23.310510	7.728652	
Cylinders	5.477707	1.699788	
Displacement	195.318471	104.331589	
Horsepower	104.869427	38.096214	
Weight	2990.251592	843.898596	
Acceleration	15.559236	2.789230	
Model Year	75.898089	3.675642	
Europe	0.178344	0.383413	
Japan	0.197452	0.398712	
USA	0.624204	0.485101	

We need to normalize our data so different data doesn't affect our model differently. If the numbers are on different scales, when they're multiplied to calculate their weight, the weightage will be totally wrong, so we have to normalize the data.

We can simply do that the we normalizer built into TensorFlow. We can do that with the following code:

```
normalizer = tf.keras.layers.Normalization(axis=-1)
```

Now that we have built our normalizer for our data, we need to apply it to our training data. We can do that with the following code:

```
normalizer.adapt(np.array(train_features))
```

We can visualize this change with the following code:

```
print(normalizer.mean.numpy())
```

The output looks like this:

```
[[ 5.478 195.318 104.869
 2990.252 15.559 75.898 0.178 0.197
 0.624]]
```

As you can see, the data has been transformed and normalized.

We can look at the changes that were made with the following code:

```
first = np.array(train_features[:1])

with np.printoptions(precision=2, suppress=True):
    print('First example:', first)
    print()
    print('Normalized:', normalizer(first).numpy())
```

The output looks like this:

```
First example:
[[ 4.  90.  75. 2125.  14.5  74.  0.  0.  1. ]]
```

```
Normalized: [[-0.87 -1.01 -0.79 -1.03 -0.38 -0.52 -0.47  
-0.5  0.78]]
```

## **Linear Regression:**

We will be using linear regression to find patterns and similarities between the variables and MPG. To start, we will be using one variable linear regression.

We will start by trying to predict MPG from Horsepower.

We will have to go through two steps:

1. Normalize the horsepower vector using `tf.keras.layers.Normalization`
2. Use a linear transformation ( $y = mx+b$ ) to make 1 output through a linear layer. We will do this using `tf.keras.layers.dense`

We can set the number of inputs through the `input_shape` argument.. First, we will create an array for Horsepower using `numpy`.

Then use the normalization step we talked about earlier

We can do that with the following code:

```
horsepower = np.array(train_features['Horsepower'])

horsepower_normalizer =
layers.Normalization(input_shape=[1,], axis=None)
horsepower_normalizer.adapt(horsepower)
```

We can now also build the Keras sequential model.

This can be done with the following code:

```
horsepower_model = tf.keras.Sequential([
horsepower_normalizer,
layers.Dense(units=1)
])

horsepower_model.summary()
```

The output looks like this:

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
normalization_1 (Normalizati	(None, 1)	3
=====		
dense (Dense)	(None, 1)	2
=====		

Total params: 5  
Trainable params: 2  
Non-trainable params: 3

---

As said before, this model will predict MPG from Horsepower. We can train our model using the first 10 Horsepower data points with the following code:

```
horsepower_model.predict(horsepower[:10])
```

The output looks like this:

```
array([[ 0.383],  
       [ 0.216],  
       [-0.707],  
       [ 0.537],  
       [ 0.486],  
       [ 0.191],  
       [ 0.576],  
       [ 0.486],  
       [ 0.127],  
       [ 0.216]], dtype=float32)
```

After our model is built, we need to set it up our training procedures and configure the loss and optimizer.

We can do that with the following code:

```
horsepower_model.compile(  
optimizer=tf.optimizers.Adam(learning_rate=0.1),  
loss='mean_absolute_error')
```

After we've configured our training procedures, we can train our model for 100 epochs with the following code:

```
%%time  
history = horsepower_model.fit(  
    train_features['Horsepower'],  
    train_labels,  
    epochs=100,  
    # Suppress logging.  
    verbose=0,  
    # Calculate validation results on 20% of the training data.  
    validation_split = 0.2)
```

The output looks like this:

```
CPU times: user 3.73 s, sys: 738 ms, total: 4.47 s  
Wall time: 2.84 s
```



We can visualize the progress of the training model with the following code:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

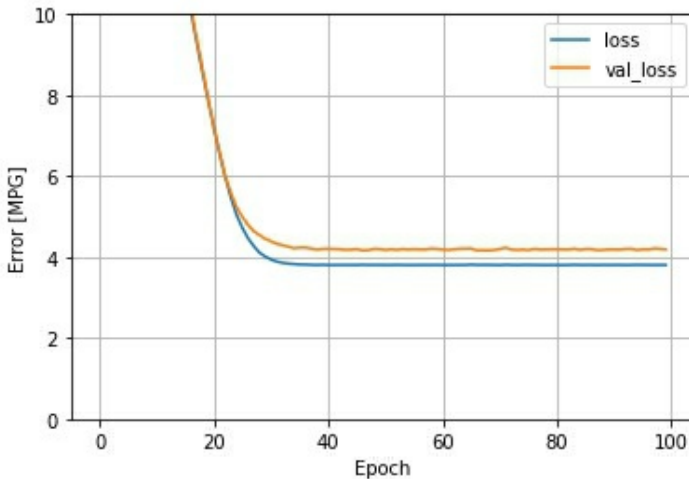
	loss	val_loss	epoch
<b>95</b>	3.806118	4.191990	95
<b>96</b>	3.804530	4.188096	96
<b>97</b>	3.802698	4.211971	97
<b>98</b>	3.805086	4.203719	98
<b>99</b>	3.803676	4.188722	99

We can plot the progress of the training model with the following code:

```
def plot_loss(history):
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.ylim([0, 10])
    plt.xlabel('Epoch')
    plt.ylabel('Error [MPG]')
    plt.legend()
    plt.grid(True)
```

```
plot_loss(history)
```

The output looks like this:



We can collect the results of the training with the following code:

```
test_results = {}  
  
test_results['horsepower_model'] =  
horsepower_model.evaluate(  
    test_features['Horsepower'],  
    test_labels, verbose=0)
```

Because we are just dealing with single variable linear regression right now, it's much easier to visualize the model's predictions.

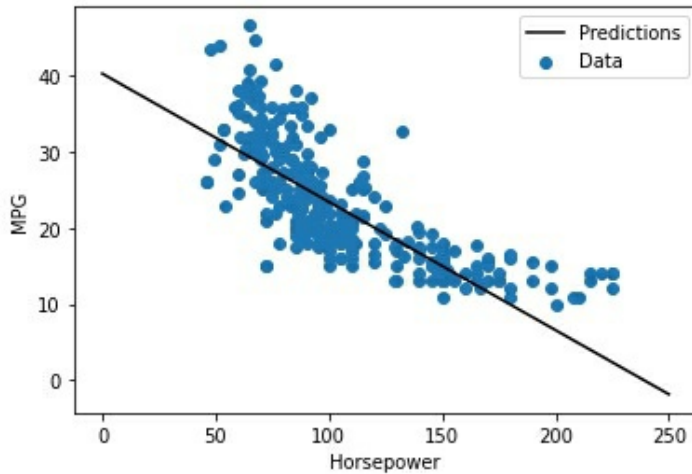
We can do that with the following code:

```
x = tf.linspace(0.0, 250, 251)
y = horsepower_model.predict(x)
```

```
def plot_horsepower(x, y):
    plt.scatter(train_features['Horsepower'], train_labels,
label='Data')
    plt.plot(x, y, color='k', label='Predictions')
    plt.xlabel('Horsepower')
    plt.ylabel('MPG')
    plt.legend()
```

```
plot_horsepower(x,y)
```

The output looks like this:



As you can see, linear regression has been applied to the data for predicting MPG.

## Linear Regression with Multiple Inputs:

Linear regression with multiple inputs is somewhat similar to just one input linear regression. We can still use the basic  $y = mx+b$  formula, but instead  $m$  will be a matrix and  $b$  will be a vector.

We will once again be using the two-step system for linear regression. The first layer will be the normalizer that we created earlier. We can use it with the following code:

If we use `model.predict` to predict values, we get only 1 output for each example. We can see this with the following code:

```
linear_model.predict(train_features[:10])
```

The output looks like this:

```
array([[ -1.466],  
       [ -0.577],  
       [  0.188],  
       [ -0.129],  
       [  1.447],  
       [  0.326],  
       [  1.413],  
       [  0.319],
```

```
[-0.425],  
[ 0.177]], dtype=float32)
```

When we want to use our model, the weights for the model will already be built in. We can see them using the following code:

```
linear_model.layers[1].kernel
```

The output looks like this:

```
<tf.Variable 'dense_1/kernel:0' shape=(9, 1)  
dtype=float32, numpy=  
array([[ 0.569],  
       [ 0.291],  
       [-0.539],  
       [ 0.373],  
       [-0.113],  
       [ 0.52 ],  
       [-0.011],  
       [ 0.625],  
       [-0.239]], dtype=float32)>
```

We can train our model for the multiple inputs using the following code:

```
linear_model.compile(  
optimizer=tf.optimizers.Adam(learning_rate=0.1),
```

```
loss='mean_absolute_error')
```

```
%%time  
history = linear_model.fit(  
    train_features,  
    train_labels,  
    epochs=100,  
    # Suppress logging.  
    verbose=0,  
    # Calculate validation results on 20% of the training data.  
    validation_split = 0.2)
```

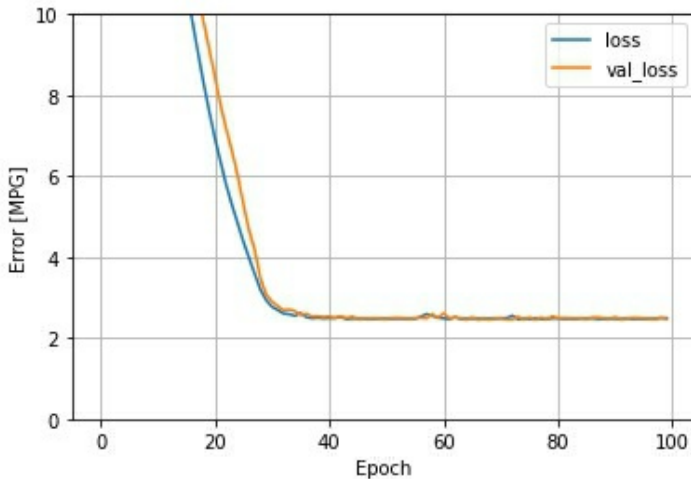
The output looks like this:

```
CPU times: user 3.96 s, sys: 721 ms, total: 4.68 s  
Wall time: 2.95 s
```

Because we used multiple inputs instead of just the single horsepower input, our loss will be much less with the multiple inputs as the model has more data to work with. We can visualize the loss with the following code:

```
plot_loss(history)
```

The output looks like this:



We can collect our data for later with the following code:

```
test_results['linear_model'] = linear_model.evaluate(
    test_features, test_labels, verbose=0)
```

We will be taking our regression even further with a Deep Neural Network (DNN).

Before, we did single input and multiple input linear regression. While this is good, the losses were pretty high as not all datapoints were linear. We will be taking a better approach to regression through DNNs.



We will be making single input and multiple input DNNs.

Because DNNs are more complex, we will have a few more layers:

- Normalization layer like before
- Two hidden layers that are non linear
- A linear single output layer

The two models that we will be making use the same compiler, so we can make it with the following code:

```
def build_and_compile_model(norm):  
model = keras.Sequential([  
    norm,  
    layers.Dense(64, activation='relu'),  
    layers.Dense(64, activation='relu'),  
    layers.Dense(1)  
)  
  
    model.compile(loss='mean_absolute_error',  
                  optimizer=tf.keras.optimizers.Adam(0.001))  
return model
```

We will now be creating our single input

DNN.

Just like the single input linear model we made, we will be using Horsepower for our input.

We can do that with the following code:

```
dnn_horsepower_model =  
build_and_compile_model(horsepower_normalizer)
```

There's more parameters than before, so we can see what some of them are with the following code:

```
dnn_horsepower_model.summary()
```

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
=====		
normalization_1 (Normalizati	(None, 1)	3
dense_2 (Dense)	(None, 64)	128
dense_3 (Dense)	(None, 64)	4160
dense_4 (Dense)	(None, 1)	65
=====		
Total params: 4,356		

Trainable params: 4,353

Non-trainable params: 3

---

We are now going to train our model with the following code:

```
%%time
history = dnn_horsepower_model.fit(
    train_features['Horsepower'],
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=100)
```

The output looks like this:

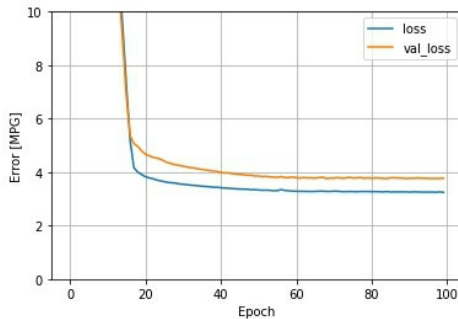
CPU times: user 4.2 s, sys: 683 ms, total: 4.88 s

Wall time: 3.24 s

We can observe the loss with the following code:

```
plot_loss(history)
```

The output looks like this:



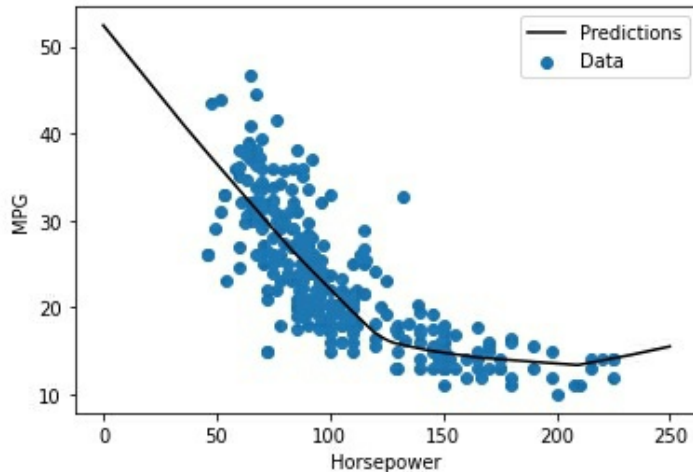
The loss was less than the loss from the linear model, but only slightly less.

Let's now plot the prediction of the functions. We can do that with the following code:

```
x = tf.linspace(0.0, 250, 251)
y = dnn_horsepower_model.predict(x)
```

```
plot_horsepower(x, y)
```

The output looks like this:



The model takes advantage of not being linear and as you can see, it matches the data better than the linear model.

We can once again collect the data for later:

```
test_results['dnn_horsepower_model'] =  
dnn_horsepower_model.evaluate(  
    test_features['Horsepower'], test_labels,  
    verbose=0)
```

## **Regression Using a DNN and Multiple Inputs:**

We will now repeat what we did with one

input but with multiple inputs now.

We can build the model with the following code:

```
dnn_model = build_and_compile_model(normalizer)
dnn_model.summary()
```

The output looks like this:

Model: "sequential_3"		
Layer (type)	Output Shape	Param #
=====		
normalization (Normalization (None, 9))	(None, 19)	
dense_5 (Dense)	(None, 64)	640
dense_6 (Dense)	(None, 64)	4160
dense_7 (Dense)	(None, 1)	65
=====		
Total params: 4,884		
Trainable params: 4,865		
Non-trainable params: 19		

We can train our model with the following code:

```
%%time
history = dnn_model.fit(
    train_features,
    train_labels,
```

```
validation_split=0.2,  
verbose=0, epochs=100)
```

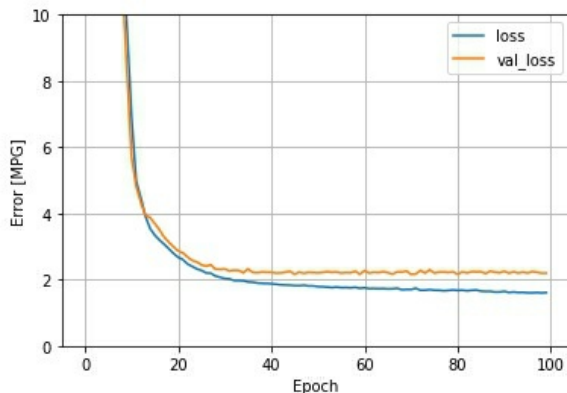
The output looks like this:

```
CPU times: user 4.1 s, sys: 723 ms, total: 4.82 s  
Wall time: 3.13 s
```

We can now look at the loss with the following code:

```
plot_loss(history)
```

The output looks like this:



The loss is even less than before meaning that the multiple inputs helped train the model better than the previous attempts we made.

We can collect the test results once again with the following code:

```
test_results['dnn_model'] =  
dnn_model.evaluate(test_features, test_labels, verbose=0)
```

We can review the test set performance with the following code:

```
pd.DataFrame(test_results, index=['Mean absolute error  
[MPG]']).T
```

The output looks like this:

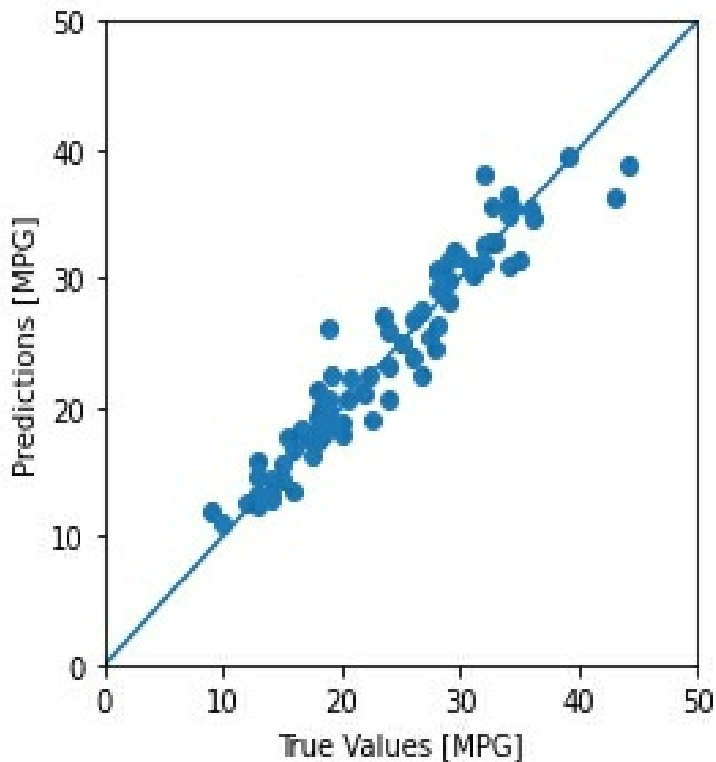
	Mean absolute error [MPG]
horsepower_model	3.638800
linear_model	2.522141
dnn_horsepower_model	2.884843
dnn_model	1.734838

We can now make predictions on the test set with the following code:



```
test_predictions =  
dnn_model.predict(test_features).flatten()  
  
a = plt.axes(aspect='equal')  
plt.scatter(test_labels, test_predictions)  
plt.xlabel("True Values [MPG]")  
plt.ylabel('Predictions [MPG]')  
lims = [0, 50]  
plt.xlim(lims)  
plt.ylim(lims)  
_ = plt.plot(lims, lims)
```

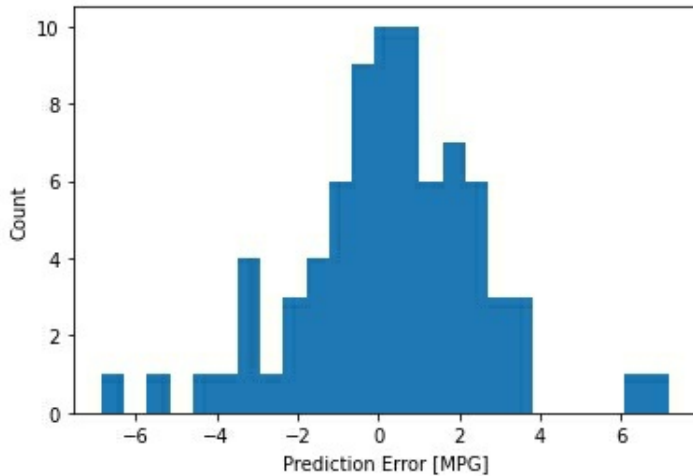
The output looks like this:



The model seems to perform pretty well. We can check the error distribution with the following code:

```
error = test_predictions - test_labels
plt.hist(error, bins=25)
plt.xlabel('Prediction Error [MPG]')
_ = plt.ylabel('Count')
```

The output looks like this:



Now that the model works, we can save it for later use with the following code:

```
dnn_model.save('dnn_model')
```

If we now reload the model with the following code, we get the exact same result:

```
reloaded = tf.keras.models.load_model('dnn_model')
```

```
test_results['reloaded'] = reloaded.evaluate(  
test_features, test_labels, verbose=0)
```

```
pd.DataFrame(test_results, index=['Mean absolute error  
[MPG]']).T
```

The output looks like this;

Mean absolute error [MPG]	
horsepower_model	3.636316
linear_model	2.443556
dnn_horsepower_model	2.970807
dnn_model	1.722730
reloaded	1.722730

With that, our last project, project 4 comes to an end!

## Closing Remarks/Further Projects

You have gone through 4 projects now and done the basics of machine learning. With the many twists and turns that exist in programming, you held on and arrived at the destination!

I wish you great on your journey ahead as a programmer and scientist. To help you further your knowledge, I have included a few links you can go to. I have not developed any of these, but they are third party resources that I found useful.

Coursera professional certificate on Tensorflow:

<https://www.coursera.org/professional-certificates/tensorflow-in-practice>

Replit.com – an online coding platform similar to google Colab that has a lot of languages

<https://replit.com/>

Codecademy machine learning

<https://www.codecademy.com/catalog/subject/machine-learning>