Introduction to Keras

In this notebook, we'll explore the Keras and use it to create a classifier to predict hand written digits.

Introduction and Background

What is Keras? Why use it?

Keras is a high-level deep learning API that serves to build, train, evaluate, and as a toolbox to neural networks. This means that Keras is used to do deep learning by being a wrapper to more complex and low-level deep learning frameworks like TensorFlow, CNTK, and Theano. The same functions for Keras wrapped CNTK are the same for Keras wrapped Theano and TensorFlow. TensorFlow (TF), the current most popular deep learning library and open sourced by Google, has recently integrated it's framework with Keras, so now Keras comes in with TF.

Well, that's all find and dandy, but why should we use Keras? What's wrong with using TF? And why is TF integrating Keras? What does this all mean? TF, as well as other deep learning libraries, can be a bit more complicated to use (for the exception of PyTorch, more on that in a second). As you may have already experienced, deep learning can be quite complex, having a *deep* understanding (pun intended) of everything that's going on is sometimes needed, making TF (pre-Keras integration) unfriendly. With the latest version of Keras, TF is now super easy to use. Since TF is the most common library, there are tons of add ons and pipelines for mobile and websites already made. Therefore, Keras/TF is ideal for the more common programmer.

Are there other options? The other up-and-coming library is PyTorch, created by Facebook. PyTorch is more common in research circles (whereas Keras/TF is more common in industry) however the interface at a high level tends to be the same. Arguably, transitioning from Keras to PyTorch should be simple.

So am I going to be using Keras or TF in this class? Both! In our case, they're the same thing. Again, Keras has been integrated by default to TF, Keras is a wrapper to TF. If you're going to be doing high level work (which will be 95% of the time), you're going to be calling and using Keras. If you're going into the nitty gritty low-level work of Neural Networks (i.e. creating your own custom loss function, activation function, or metrics), you're going to be using TF. If someone asks what deep learning framework you used in this class, TensorFlow! But do give a shoutout to Keras if you use `tf.keras` often.

So for the rest of this notebook, we'll be using Keras.

Training Deep Learning Networks

Neural nets can take a while to train; you may have heard this prior in the mystery of deep learning. So can your computer train neural networks? It depends. If you're doing **deep** deep learning, using +30 layers or working with an enormous amount of data, training could take forever. One would need a GPU on their computer. More than often, we use some sort of cloud computing, why Colab would be ideal. For more intensive trainings (since Colab shuts down after +12 hours), usually a cloud provider like AWS (Amazon) or GCP (Google) would be used.

Keras/TF works with and without a GPU, but by default is set to without a GPU. In this notebook, we won't be working with intense computation, so a GPU isn't necessarily. However that doesn't mean that training can't take a while. If you're working on Colab, changing the computation setting from CPU to GPU may speed things up. Your decision on what to do.

If you have a GPU on your computer, there are guides towards setting up TF with a GPU with certain requirements. Slack me (Kevin Marroquin) for more details since this installation can get tricky.

Running an example: MNIST

Installing Tensorflow and Loading Imports

First, we're going to install Keras, in case you're using this on your local computer. However, this should work under Colab.

```
#See installation details here: https://www.tensorflow.org/install/
!pip install tensorflow
```

```
    Requirement already satisfied: tensorflow in /usr/local/lib/python3.6/dist-packa
    Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.6/dis
    Requirement already satisfied: h5py<2.11.0,>=2.10.0 in /usr/local/lib/python3.6/
    Requirement already satisfied: wrapt>=1.11.1 in /usr/local/lib/python3.6/dist-pa
    Requirement already satisfied: grpcio>=1.8.6 in /usr/local/lib/python3.6/dist-pa
    Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.6/dist-
    Requirement already satisfied: tensorflow-estimator<2.4.0,>=2.3.0 in /usr/local/
    Requirement already satisfied: numpy<1.19.0,>=1.16.0 in /usr/local/lib/python3.6
    Requirement already satisfied: google-pasta>=0.1.8 in /usr/local/lib/python3.6/d
    Requirement already satisfied: keras-preprocessing<1.2,>=1.1.1 in /usr/local/lib
    Requirement already satisfied: tensorboard<3,>=2.3.0 in /usr/local/lib/python3.6
    Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.6/dist-pack
    Requirement already satisfied: gast==0.3.3 in /usr/local/lib/python3.6/dist-pack
    Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.6/dist
    Requirement already satisfied: astunparse==1.6.3 in /usr/local/lib/python3.6/dis
    Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.6/dist-pack
    Requirement already satisfied: scipy==1.4.1 in /usr/local/lib/python3.6/dist-pac
    Requirement already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.6/dist-p
    Requirement already satisfied: setuptools in /usr/local/lib/python3.6/dist-packa
    Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/p
    Requirement already satisfied: google-auth<2,>=1.6.3 in /usr/local/lib/python3.6
    Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/li
    Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.6/dist-
    Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.6/d
    Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.6/dis
    Requirement already satisfied: rsa<5,>=3.1.4; python_version >= "3" in /usr/loca
    Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.6
    Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.
    Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python
    Requirement already satisfied: importlib-metadata; python_version < "3.8" in /us
    Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-pac
    Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/l
    Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/di
    Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dis
    Requirement already satisfied: pyasn1>=0.1.3 in /usr/local/lib/python3.6/dist-pa
    Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.6/dist-
    Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.6/dist-packag
```

```
#Importing Keras and other imports
from tensorflow import keras
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

We're going to want reproducible results, so pre-defining random seeds is necessary.

```
from numpy.random import seed
seed(42)
from tensorflow.random import set_seed
set_seed(42)
#If the two lines above give you error, mute them and run the following:
# import tensorflow
# tensorflow.random.set_seed(42)
```

Normally one wouldn't want to have a pre-defined random seed (in fact you would want the exact opposite), but in our case, since we want to be able to produce reprodicible results for grading purposes, it's needed.

Loading, Analyzing, and Preprocessing Data

For this example, we will be using the MNIST dataset. MNIST is a dataset of handwritten digits created to classify what type of digit, from zero to nine, is shown. MNIST is modified and, to some extent, normalized such that they're all the same size. Let's explore and see what kind of information MNIST have. If you've seen MNIST, this will probably be stuff you've seen before, so feel free to skip to the neural network part.

```
#Loading datasets
mnist = keras.datasets.mnist
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

```
#Seeing shapes of data
print("X_train:", X_train_full.shape)
print("y_train:", y_train_full.shape)
print("X_test:", X_test.shape)
print("y_test:", y_test.shape)
```

```
X_train: (60000, 28, 28)
y_train: (60000,)
X_test: (10000, 28, 28)
y_test: (10000,)
```

So we see that we have 60000 training datapoints and 10000 test. How does one of these datapoints look like?

```
X_train_full
```

```
array([[[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       ...,

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]]], dtype=uint8)
```

```
y_test[0]
```

```
7
```

```
X_train_full[0].shape
```

```
(28, 28)
```

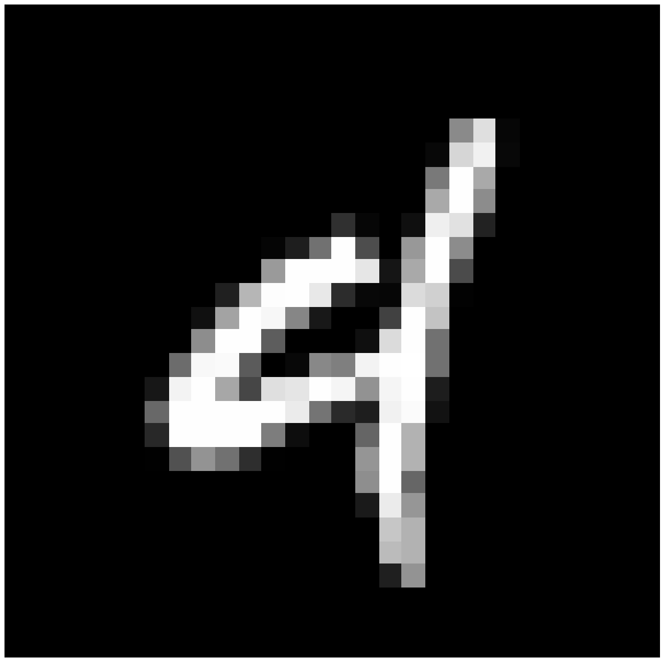
So an image is represented as a 28x28 image in pixels and it. It's a picture in black and white, representing pixel intensity from 0 to 255, where a 255 representing black and 0 representing white. Let's plot this to see the data.

```
#Looking at test data
y_train_full
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

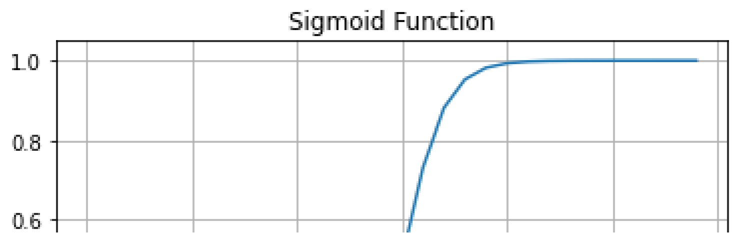
```
#Rerun to get different numbers
def plotImage(image):
    """A 28x28 array that represents an image."""
    plt.figure(figsize=(6, 6))
    plt.imshow(image, cmap="gray")
    plt.axis(False)
    plt.show()
sample_image = np.random.randint(0, len(X_train_full))
print("The expected value is:", y_train_full[sample_image])
plotImage(X_train_full[sample_image])
```

```
The expected value is: 4
```



Before using this data, we must preprocess it to our likings. Something that should be done before plugging our data into our neural net is to normalize and shift it for better results ([see more details here](#)). If you want some quick intuition about our activation functions, or a sigmoid function, most of it's "function" (the part that's interesting and not a flat line) is near `x` = 0 than at `x` = 255, `X_train_full`'s max point. Normalizing and shifting our data brings it to the action of the function (seriously, no pun intended on this one) where the model may perform better.

```
#Example of Sigmoid function
def sigmoid(x):
    return 1/(np.exp(-x) + 1)
plt.plot(np.arange(-15, 15), sigmoid(np.arange(-15, 15)))
plt.title("Sigmoid Function")
plt.grid()
plt.show()
```



Let's convert this data to 0 and 1, where 1 is the max pixel intensity (255) and zero is still zero. Anything in between is the scaled value of the original data. In the process, we will create a validation pair (of X and y) to test our model after training and testing.

```
X_valid, X_train = X_train_full[:5000] / 255, X_train_full[5000:] / 255
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255
#No need to do y_test since it's not being fed into the data.
#Also, data is already shuffled, so no need to do so here
```

Lastly, inputting a 2D datapoint into a neural network is a bit of work to implement and architect. To avoid harder work, we will transform our data into a 1D array. We will do this in the next section.

Creating A Simple Neural Network

Let's build a basic 2-layered neural network.

```
#Including random seeds here again in case for reproducibility
from numpy.random import seed
seed(42)
from tensorflow.random import set_seed
set_seed(42)
#If the two lines above give you error, mute them and run the following:
# import tensorflow
# tensorflow.random.set_seed(42)

model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="sigmoid"))
model.add(keras.layers.Dense(100, activation="sigmoid"))
model.add(keras.layers.Dense(10, activation="softmax"))

#Another similar way of writing the above code is:
# model = keras.model.Sequential([
#     keras.layers.Flatten(input_shape=[28, 28]),
#     keras.layers.Dense(300, activation="sigmoid"),
#     keras.layers.Dense(100, activation="sigmoid"),
#     keras.layers.Dense(10, activation="softmax")
# ])
```

```
" ])
#Your choice in preference.
```

Going through the above code line by line:

- The first line creates the simplest model composed of a single stack of layers connected sequentially. Needed in most cases to initialize the model.
- The next layer flattens the data from 2D 28x28 to 1D 784. This layer is known as the input layer.
- The next two (Dense) layers are hidden layers with 300 and 100 neurons respectively. They're both using sigmoid for their activation functions and contain a weight matrix that changes during training.
- The last dense layer is the output layer. Choosing 10 neurons is a reflection of our output. We use a sigmoid function in order to calculate the probability.

This is, again, a 2-layered neural network. Let's print out the summary of the model.

```
#Inputting the first hidden layer, should be in the shape of 0-1
weight = model.layers[1].get_weights()
weight[0]
```

```
    array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
             0.03859074, -0.06889391],
           [ 0.00476504, -0.03105379, -0.0586676 , ...,  0.00602964,
            -0.02763776, -0.04165364],
           [-0.06189284, -0.06901957,  0.07102345, ..., -0.04238207,
             0.07121518, -0.07331658],
           ...,
           [-0.03048757,  0.02155137, -0.05400612, ..., -0.00113463,
             0.00228987,  0.05581069],
           [ 0.07061854, -0.06960931,  0.07038955, ..., -0.00384101,
             0.00034875,  0.02878492],
           [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
             0.00272203, -0.06793761]], dtype=float32)
```

We mentioned weights, that each layer contains it's own weights. As a reminder, a weight is needed to put emphasis on a feature in order to help a model model.

We'll now be compiling the model

```
model.compile(loss="sparse_categorical_crossentropy", #Loss function
              optimizer="sgd", #Stochastic Gradient Descent
             metrics=["accuracy"])
```

Let's train and evaluate the model.

```
y_valid.shape
```

```
(5000,)
```

```
X_valid.shape
```

```
(5000, 28, 28)
```

```
X_train.shape
```

```
(55000, 28, 28)
```

```
y_train.shape
```

```
(55000,)
```

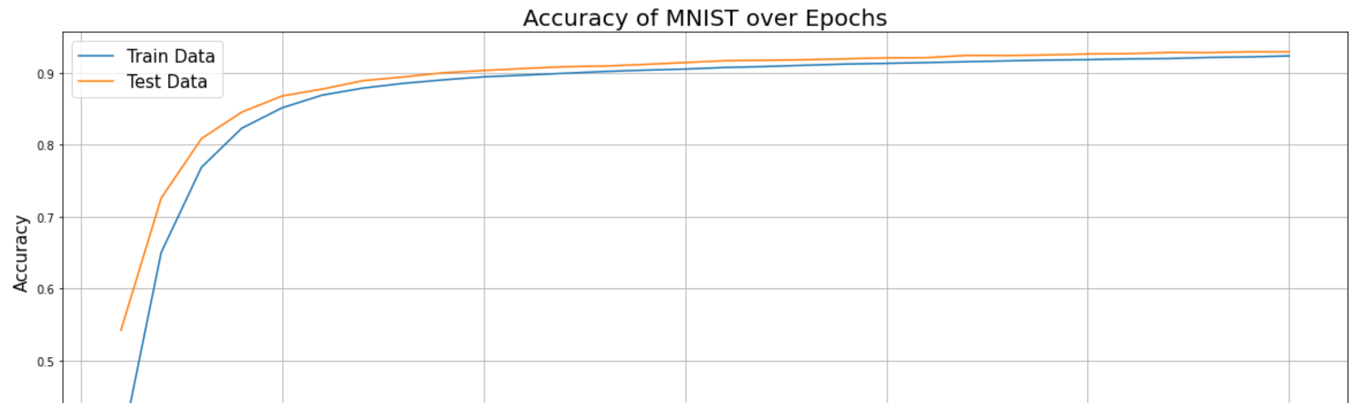
```
#May take a while
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/30
1719/1719 [==============================] - 5s 3ms/step - loss: 2.1718 - accu
Epoch 2/30
1719/1719 [==============================] - 5s 3ms/step - loss: 1.5916 - accu
Epoch 3/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.9993 - accu
Epoch 4/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.7290 - accu
Epoch 5/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.5895 - accu
Epoch 6/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.5084 - accu
Epoch 7/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4572 - accu
Epoch 8/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4227 - accu
Epoch 9/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3980 - accu
Epoch 10/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3796 - accu
Epoch 11/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3652 - accu
Epoch 12/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3534 - accu
Epoch 13/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3436 - accu
Epoch 14/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3353 - accu
Epoch 15/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3279 - accu
Epoch 16/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3212 - accu
Epoch 17/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3156 - accu
Epoch 18/30
```

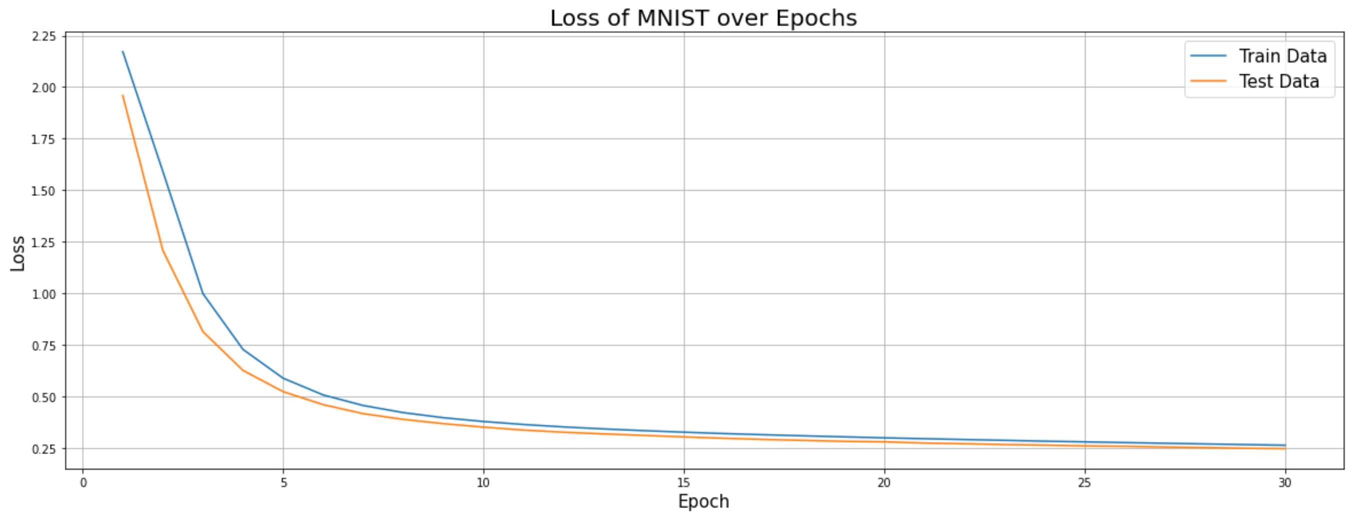
```
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3102 - accu
Epoch 19/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3053 - accu
Epoch 20/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3004 - accu
Epoch 21/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2963 - accu
Epoch 22/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2922 - accu
Epoch 23/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2884 - accu
Epoch 24/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2844 - accu
Epoch 25/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2810 - accu
Epoch 26/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2775 - accu
Epoch 27/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2742 - accu
Epoch 28/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2709 - accu
Epoch 29/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2678 - accu
Epoch 30/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2647 - accu
```

Plotting the accuracy and loss of our training and validation data.

```
#Plotting Accuracy
plt.figure(figsize = (20, 7))
plt.plot(np.arange(1, 31), history.history["accuracy"], label = "Train Data")
plt.plot(np.arange(1, 31), history.history["val_accuracy"], label = "Test Data")
plt.legend(fontsize = 15)
plt.xlabel("Epoch", size = 15)
plt.ylabel("Accuracy", size = 15)
plt.title("Accuracy of MNIST over Epochs", size = 20)
plt.grid()
plt.show()
```



```
#Plotting Loss
plt.figure(figsize = (20, 7))
plt.plot(np.arange(1, 31), history.history["loss"], label = "Train Data")
plt.plot(np.arange(1, 31), history.history["val_loss"], label = "Test Data")
plt.legend(fontsize = 15)
plt.xlabel("Epoch", size = 15)
plt.ylabel("Loss", size = 15)
plt.title("Loss of MNIST over Epochs", size = 20)
plt.grid()
plt.show()
```



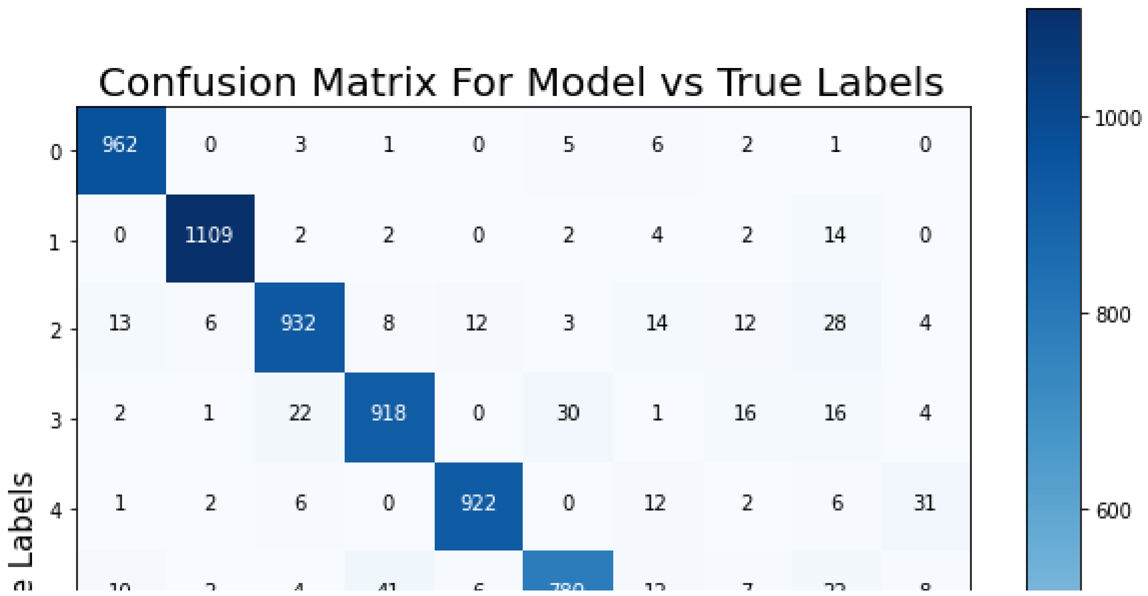
Since our training and validation data seems to be closely tied, we can conclude there's hardly any overfitting. Creating a confusion matrix

```
#Determining error from confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, np.argmax(model.predict(X_test), axis = -1))
```

```
    array([[ 962,    0,    3,    1,    0,    5,    6,    2,    1,    0],
           [   0, 1109,    2,    2,    0,    2,    4,    2,   14,    0],
           [  13,    6,  932,    8,   12,    3,   14,   12,   28,    4],
           [   2,    1,   22,  918,    0,   30,    1,   16,   16,    4],
           [   1,    2,    6,    0,  922,    0,   12,    2,    6,   31],
           [  10,    2,    4,   41,    6,  780,   12,    7,   22,    8],
           [  14,    3,    6,    1,   10,   16,  904,    2,    2,    0],
           [   4,    9,   22,    6,    6,    0,    0,  962,    1,   18],
           [   7,    7,    8,   24,    9,   33,   11,   12,  859,    4],
           [  13,    8,    1,   10,   33,    6,    0,   24,    6,  908]])
```

A nicer plot

```
#Creating a plotting function for a confusion matrix
import itertools
cm = confusion_matrix(y_test, np.argmax(model.predict(X_test), axis = -1))
plt.figure(figsize=(10, 10))
plt.imshow(cm, plt.cm.Blues)
plt.colorbar()
plt.title("Confusion Matrix For Model vs True Labels", size = 20)
fmt = 'd'
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
              horizontalalignment="center",
              color="white" if cm[i, j] > thresh else "black")
plt.xlabel("Predicted Labels", size = 15)
plt.xticks(np.arange(0, 10))
plt.ylabel("True Labels", size = 15)
plt.yticks(np.arange(0, 10))
plt.show()
```



A confusion matrix is useful in looking at what values are predicting expected values. For example, the graph above shows that our model predicted nine +30 times when in reality it was four. This checks out, as fours and nines can occasionally look alike.

A note on reading a confusion matrix: The heavy blue diagonal represents correctly predicted labels. Everything outside the diagonal represents predicted labels that were wrong to guess a certain true label. Match the x-axis and y-axis to determine what values were guessed correctly/incorrectly. What's the highest incorrectly predicted number? Does it make sense why a model would confuse it (i.e. can the incorrectly predicted value and true label be similarly drawn)? Would a human confuse this incorrectly predicted number?

Finally, we see how our model performs using our test data.

```
score = model.evaluate(X_test, y_test)
print("loss:", score[0], "acc:", score[1])
```

```
    313/313 [==============================] - 0s 2ms/step - loss: 0.2563 - accuracy
    loss: 0.25626498460769653 acc: 0.925599992275238
```

Question 1: Your Turn, Get Better Results Than Me On MNIST

Can you get better accuracy than me? I know there's specific hyperparameters where accuracy can get +99%. Here's a list of some hyperparameters you can tune:

Hyperparameters:

- Number of layers
- Number of neurons in layers
- Activation functions
- Learning rate
- Loss
- Optimizer
- Metrics
- Epochs

Hyperparameter testing shows how expensive testing every single option can be. I recommend changing the learning rate, number of layers, number of neurons, and activation function, but feel free to change or add whatever you'd like.

Cheatsheet on how to do the above:

- **Adding a layer**: `model.add(keras.layers.Dense())`
- **Number of neurons in layers**: Input/change number inside `keras.layers.Dense()`
- **Changing Activation functions**: In a dense layer, write `Dense(activation="InsertFunctionNameHere")`. Examples of other functions can be found here
- **Changing learning rate**: Involves importing `backend`. Instructions/a quick example can be found here.
- **Changing Loss functions**: Same as changing Activation functions, but the loss is defined/changed when you call `model.compile(loss="InsertFunctionNameHere")`. Be mindful to see what losses are appropriate for what models (you don't want a continuous loss on a binary model; continous and discrete don't mix!). This shouldn't be tweaked too much, but I want to give you all full control. List of loss functions here.
- **Changing Optimizer**: This can speed up or give you better results: `model.compile(optimizer="InsertOptimizerNameHere")`. List of Optimizers are here. Try Adam or RMSprop since they seem to be popular.
- **Changing or Adding Metrics**: Similar to changing optimizer and loss: `model.compile(metrics=["InsertListOfMetricsHere"])`. You can run multiple metrics at the same time and it shouldn't decrease your performance. The same advise goes here as to changing loss functions: be aware what metrics are telling you what before you go on and change them. Here's a list of metrics.
- **Changing Number of Epochs**: Changed when running `model.fit(epochs="InsertNumericValueHere")`. Strategies of running a certain number of epochs can be found here.

Another way to improve results is to do image processing on the data. Since this notebook is to practice Keras I won't give any hints. But I do want to point it out since in real life, modifying data is universal to improving and speeding any model. You're more likely to do this first as a strategy than tweak for model parameters. For now, however, we do this for practice

```
# from numpy.random import seed
# seed(42)
# from tensorflow.random import set_seed
# set_seed(42)
#If the two lines above give you error, mute them and run the following:
import tensorflow
tensorflow.random.set_seed(42)
```

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
```

```
#TODO: Create model, compile, and train (and I highly recommend that you
#plot/make a confusion matrix, as it's super helpful in gaining context to
#how your model is acting and where is error largest.)
###YOUR CODE BELOW###
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")



# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
    x_train shape: (60000, 28, 28, 1)
    60000 train samples
    10000 test samples
```

```
###EXTRA CELL IF NEEDED###
#Feel free to create as many as needed.
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 1600)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1600)              0
_________________________________________________________________
dense_10 (Dense)             (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

```
batch_size = 128
epochs = 15

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"]

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.
```

```
Epoch 1/15
422/422 [==============================] - 40s 94ms/step - loss: 0.3697 - accura
Epoch 2/15
422/422 [==============================] - 39s 93ms/step - loss: 0.1133 - accura
Epoch 3/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0853 - accura
Epoch 4/15
422/422 [==============================] - 39s 94ms/step - loss: 0.0728 - accura
Epoch 5/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0627 - accura
Epoch 6/15
```

```
422/422 [==============================] - 40s 95ms/step - loss: 0.0570 - accura
Epoch 7/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0522 - accura
Epoch 8/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0466 - accura
Epoch 9/15
422/422 [==============================] - 39s 94ms/step - loss: 0.0444 - accura
Epoch 10/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0423 - accura
Epoch 11/15
422/422 [==============================] - 39s 93ms/step - loss: 0.0398 - accura
Epoch 12/15
422/422 [==============================] - 39s 93ms/step - loss: 0.0386 - accura
Epoch 13/15
422/422 [==============================] - 39s 93ms/step - loss: 0.0351 - accura
Epoch 14/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0352 - accura
Epoch 15/15
422/422 [==============================] - 40s 94ms/step - loss: 0.0325 - accura
<tensorflow.python.keras.callbacks.History at 0x7f5271937710>
```

```
# your_score = model.evaluate(x_test, y_test, verbose=0)
# print("Test loss:", your_score[0])
# print("Test accuracy:", your_score[1])
```

```
Test loss: 0.026523232460021973
Test accuracy: 0.9915000200271606
```

```
#Tester cell to see if your model is beating my results
#Run this for your credit
your_score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", your_score[0])
print("Test accuracy:", your_score[1])
print("OG Test loss:", score[0])
print("OG Test accuracy:", score[1])
assert your_score[1] > score[1]
```

```
Test loss: 0.026523232460021973
Test accuracy: 0.9915000200271606
OG Test loss: 0.25626498460769653
OG Test accuracy: 0.925599992275238
```

Question 2: Create a Fashion Classifier

MNIST is known as being "too easy" for machine learning models. Classical ML models (e.g. SVMs, Random Forests) are able to score +99% accuracy with less training. Another similar dataset, named *Fashion MNIST*, is gaining popularity as it's a much harder classification problem while quite similar to MNIST.

In this section, you will be creating a classifier for this fashion dataset. This dataset is very similar to MNIST in structure, so there won't be much hand holding in this section. In theory, you can copy/paste everything in the previous section and change very little. I recommend typing everything out (particularly in the Keras section) as it gets you familiar with actually learning Keras.

A note about Fashion MNIST: the dataset is heavily pre-processed. The data won't look pretty to us (it's pixalated and colorless), but to a computer it's simple enough.

```
# from numpy.random import seed
# seed(42)
# from tensorflow.random import set_seed
# set_seed(42)
#If the two lines above give you error, mute them and run the following:
import tensorflow as tf
tensorflow.random.set_seed(42)
import numpy as np
import matplotlib.pyplot as plt
```

```
#Load Data
#HINT: You can load Fashion MNIST from Keras the same way you do with MNIST.
#You may have to do some digging on the internet in order to do this.
#This exercise is to get you familiar with the Keras API
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

#Your labels in y are numbers from 0-9. Each of those numbers corresponds to
#an index to a label in the labels variable. i.e. 8 = "Bag", 4 = "Coat", etc.
labels = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal",
         "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

```
    Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
    32768/29515 [================================] - 0s 0us/step
    Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
    26427392/26421880 [==============================] - 0s 0us/step
    Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
    8192/5148 [==============================================] - 0s 0us/step
    Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
    4423680/4422102 [==============================] - 0s 0us/step
```

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

```
train_images.shape
```

```
    (60000, 28, 28)
```

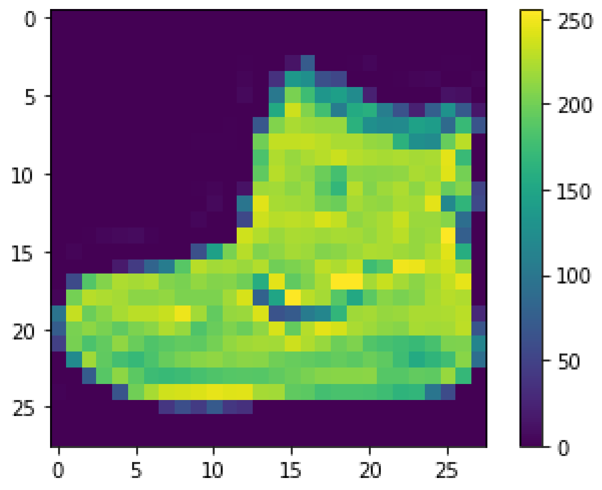
```
len(train_labels)
```

```
     60000
```

```
train_labels
```

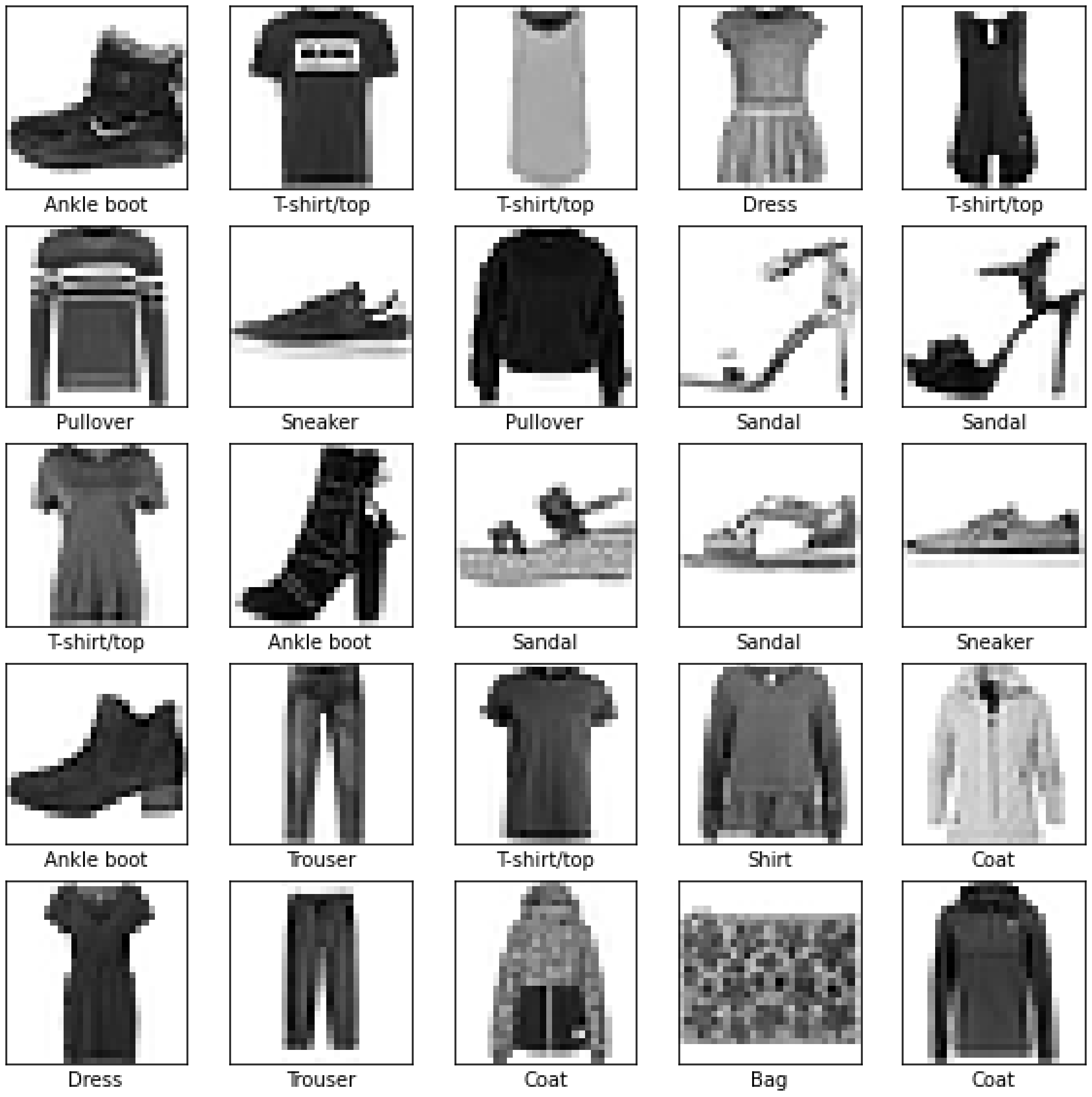
```
     array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

```
#Pre-Process Data
#Hint: Very similar to MNIST
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



```
#Create model
from numpy.random import seed
seed(42)
from tensorflow.random import set_seed
set_seed(42)
#If the two lines above give you error, mute them and run the following:
# import tensorflow
# tensorflow.random.set_seed(42)
```

```
#Plots (Optional)
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()
```



```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

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

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

```
    Epoch 1/10
    1875/1875 [==============================] - 4s 2ms/step - loss: 3.8076 - accura
    Epoch 2/10
    1875/1875 [==============================] - 4s 2ms/step - loss: 0.6881 - accura
    Epoch 3/10
    1875/1875 [==============================] - 4s 2ms/step - loss: 0.6315 - accura
```

```
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5821 - accura
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5546 - accura
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5319 - accura
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5183 - accura
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5072 - accura
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4902 - accura
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4860 - accura
<tensorflow.python.keras.callbacks.History at 0x7f52701a1f28>
```

```
#Tester cell to see if your model is beating my results
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
assert 0.89 < score[1], "Accuracy not high enough"
```

```
313/313 - 0s - loss: 0.5772 - accuracy: 0.8261

Test accuracy: 0.8260999917984009
```

Try to get above 89% accuracy for both train and test using various hyperparameters.

What label is the most mislabeled (contains the most error) in this model? Plot a confusion matrix showing this. Does it make sense? Plot several datapoints using `plotImage` to find out.

```
#Confusion Matrix
import itertools
cm = confusion_matrix(y_test, np.argmax(model.predict(X_test), axis = -1))
plt.figure(figsize=(10, 10))
plt.imshow(cm, plt.cm.Blues)
plt.colorbar()
plt.title("Confusion Matrix For Model vs True Labels", size = 20)
fmt = 'd'
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
              horizontalalignment="center",
              color="white" if cm[i, j] > thresh else "black")
plt.xlabel("Predicted Labels", size = 15)
plt.xticks(np.arange(0, 10), labels=labels, rotation = 45)
plt.ylabel("True Labels", size = 15)
plt.yticks(np.arange(0, 10), labels=labels)
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-130-4a0e3dbc60d5> in <module>()
      1 #Confusion Matrix
      2 import itertools
----> 3 cm = confusion_matrix(y_test, np.argmax(model.predict(X_test), axis =
-1))
      4 plt.figure(figsize=(10, 10))
      5 plt.imshow(cm, plt.cm.Blues)

1 frames
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py in
_check_targets(y_true, y_pred)
     88     if len(y_type) > 1:
     89         raise ValueError("Classification metrics can't handle a mix of
{0} "
---> 90                          "and {1} targets".format(type_true, type_pred))
     91 
     92     # We can't have more than one value on y_type => The set is no more
needed
```

Question 3: Bonus - EMNIST

EMNIST is an expansion of MNIST on Kaggle. If you feel you still need some practice, feel free to check it out. Others have made notebooks on Kaggle with other fancy techniques. Explore if you have time!

If you're interested in similar preprocessed datasets, check out this website: https://analyticsindiamag.com/fashion-and-medical-mnist/. They link to medical datasets and a sign language-like MNIST.

Submission

Once you're ready to submit, turn in a copy of this Jupyter notebook and a PDF copy of it. For the PDF, please label the pages of questions 1, 2, and (optionally) 3.

Locally, there should a "Download as" under "File" and PDF should be an option.

If you're doing this assignment using Colab, Colab does not have this PDF option. I recommend openning this notebook using IPython/Jupyter Notebook locally and doing the instruction above. If you're resistant to this option, there are other options. If you're on a Mac, a hack you can do is to print this page and before you print, save the file as a PDF instead of printing.

An alternative is to save your Colab notebook as an HTML file and convert that into a PDF (using a third party website like https://html2pdf.com/ or a browser extension like https://chrome.google.com/webstore/detail/web-page-to-pdfconverter/bbfoccanbdeldjaelafmbgonagegdndg). Converting HTMLs into PDFs tends to be easier/more common than ipynb to PDFs. If you need additional help, please feel free to Slack Kevin or Jess for aid!