In this tutorial, you will be using the machine learning library TensorFlow with Python3.
MNIST is a simple computer vision dataset, consisting of images of handwritten digits and labels.
Most Machine Learning researchers are familiar with this dataset as it is relatively easy to work with.

The goal of this challenge is to get familiar with tensorflow if you have not used it before
and also get used to the structure of the code we provide to help you in the competition.

If you don’t have TensorFlow installed on your system. Open a terminal and type in:

- Fedora:
  `sudo dnf install python3-dev python3-pip`
  `sudo pip3 install --user --upgrade tensorflow`

- Ubuntu:
  `sudo apt install python3-dev python3-pip`
  `sudo pip3 install --user --upgrade tensorflow`

If you don’t have numpy and matplotlib installed, you’ll need them. Open a terminal and type in:

`sudo pip3 install numpy matplotlib`

- To begin, we will paste this code into a file called train.py in

order to import the MNIST data set:

```
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt
import numpy as np
import random as ran
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

First, let's define a couple of functions that will assign the amount of training and test data we will load from the data set. It's not vital to look very deeply at these unless you want to figure out what's going on behind the scenes.

- You will need to copy and paste each function in train.py:

```
def TRAIN_SIZE(num):
print ('Total Training Images in Dataset = ' +
str(mnist.train.images.shape))
print ('-----------------------------------------------')
x_train = mnist.train.images[:num,:]
print ('x_train Examples Loaded = ' + str(x_train.shape))
y_train = mnist.train.labels[:num,:]
print ('y_train Examples Loaded = ' + str(y_train.shape))
print('')
return x_train, y_train
```

```
def TEST_SIZE(num):
print ('Total Test Examples in Dataset = ' +
```

```
str(mnist.test.images.shape))
print (’-----------------------------------------------’)
x_test = mnist.test.images[:num,:]
print ('x_test Examples Loaded = ’ + str(x_test.shape))
y_test = mnist.test.labels[:num,:]
print ('y_test Examples Loaded = ’ + str(y_test.shape))
return x_test, y_test
```

And we’ll define some simple functions for resizing and displaying the data:

```
def display_digit(num):
print(y_train[num])
label = y_train[num].argmax(axis=0)
image = x_train[num].reshape([28,28])
plt.title(‘Example: %d Label: %d’ % (num, label))
plt.imshow(image, cmap=plt.get_cmap(‘gray_r’))
plt.show()
```

```
def display_mult_flat(start, stop):
images = x_train[start].reshape([1,784])
for i in range(start+1,stop):
images = np.concatenate((images, x_train[i].reshape([1,784])))
plt.imshow(images, cmap=plt.get_cmap(‘gray_r’))
plt.show()
```

Now, we’ll get down to the business of building and training our model. First, we define variables with how many training and test examples we

would like to load.
For now, we will load all the data but we will change this value later on to save resources:

> *x_train, y_train = TRAIN_SIZE(55000)*

- Execute your code:

```
python3 train.py
```

And verify that you obtain the following result:

> *Total Training Images in Dataset = (55000, 784) -*

> *x_train Examples Loaded = (55000, 784)*
> *y_train Examples Loaded = (55000, 10)*

So, what does this mean? In our data set, there are 55,000 examples of handwritten digits from zero to nine. Each example is a 28x28 pixel image flattened in an array with 784 values representing each pixel's intensity. The examples need to be flattened for TensorFlow to make sense of the digits linearly. This shows that in x_train we have loaded 55,000 examples each with 784 pixels. Our x_train variable is a 55,000 row and 784 column matrix.

The y_train data is the associated labels for all the x_train examples. Rather than storing the label as an integer, it is stored as a 1x10 binary

array with the one representing the digit. This is also known as one-hot encoding. In the example below, the array represents a 7:

Label	0	1	2	3	4	5	6	7	8	9
Array	[0 ,	0,	0,	0,	0,	0,	0,	1,	0,	0]

So, let's pull up a random image using one of our custom functions that takes the flattened data, reshapes it, displays the example, and prints the associated label:

- Add the following line to your code:

> *display_digit(ran.randint(0, x_train.shape[0]))*

And execute it again:

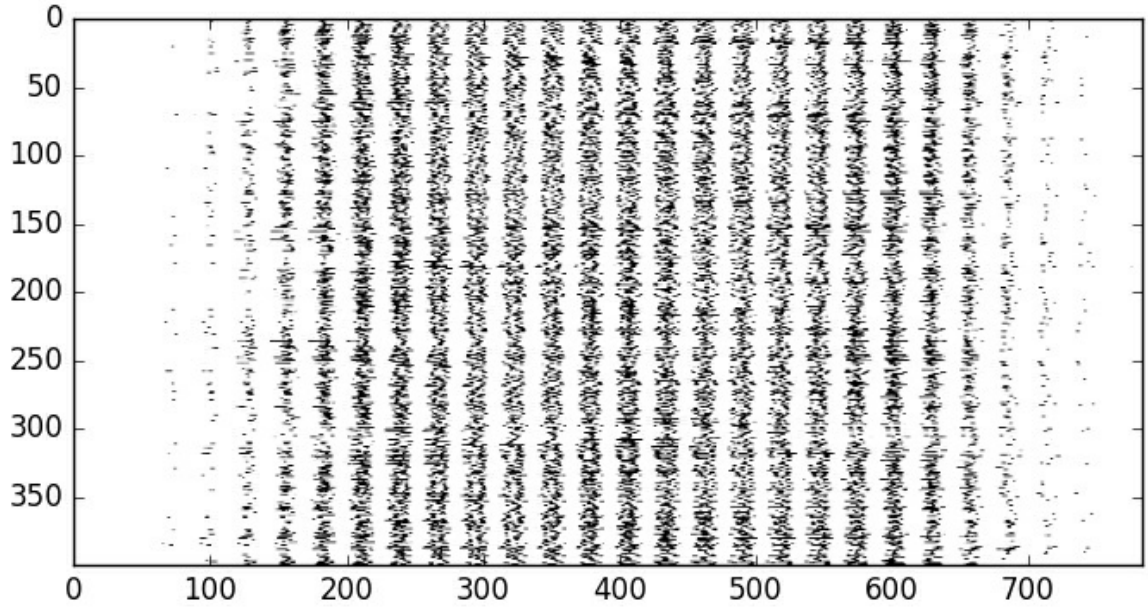
```
python3 train.py
```

Here is what multiple training examples look like to the classifier in their flattened form. Of course, instead of pixels, our classifier sees values from zero to one representing pixel intensity:

- Add the following line to your code:

> *display_mult_flat(0,400)*

Try re-executing and see how does a training set look:



Until this point, we actually have not been using TensorFlow at all. The next step is importing TensorFlow and defining our session. TensorFlow, in a sense, creates a directed acyclic graph (flow chart) which you later feed with data and run in a session:

> *import tensorflow as tf*
>
> *sess = tf.Session()*

Next, we can define a placeholder. A placeholder, as the name suggests, is a variable used to feed data into. The only requirement is that in order to feed data into this variable, we need to match its shape and type exactly. The TensorFlow website explains that “A placeholder exists solely to serve as the target of feeds. It is not initialized and contains no data.” Here, we define our x placeholder as the variable to feed our x_train data into:

```
x = tf.placeholder(tf.float32, shape=[None, 784])
```

When we assign None to our placeholder, it means the placeholder can be fed as many examples as you want to give it. In this case, our placeholder can be fed any multitude of 784-sized values.

We then define y_, which will be used to feed y_train into. This will be used later so we can compare the ground truths to our predictions. We can also think of our labels as classes:

```
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

Next, we will define the weights W and bias b. These two values are the grunt workers of the classifier—they will be the only values we will need to calculate our prediction after the classifier is trained.

We will first set our weight and bias values to zeros because TensorFlow will optimize these values later. Notice how our W is a collection of 784 values for each of the 10 classes:

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

I like to think of these weights as 10 cheat sheets for each number. This is similar to how a teacher uses a cheat sheet transparency to grade a multiple choice exam. The bias, unfortunately, is a little beyond the scope of this tutorial, but I like to think of it as a special relationship

with the weights that influences our final answer.

We will now define y, which is our classifier function. This particular classifier is also known as multinomial logistic regression. We make our prediction by multiplying each flattened digit by our weight and then adding our bias:

> *y = tf.nn.softmax(tf.matmul(x,W) + b)*

Next, we will create our cross_entropy function, also known as a loss or cost function. It measures how good (or bad) of a job we are doing at classifying. The higher the cost, the higher the level of inaccuracy. It calculates accuracy by comparing the true values from y_train to the results of our prediction y for each example. The goal is to minimize your loss:

> *cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))*

Below is where we can now assign custom variables for training if we wish. Any value that is in all caps below is designed to be changed and messed with. In fact, I encourage it! First, use these values, then later notice what happens when you use too few training examples or too high or low of a learning rate.

> *x_train, y_train = TRAIN_SIZE(5500)*
> *x_test, y_test = TEST_SIZE(10000)*
> *LEARNING_RATE = 0.1*

```
TRAIN_STEPS = 2500
```

We can now initialize all variables so that they can be used by our TensorFlow graph:

```
init = tf.global_variables_initializer()
sess.run(init)
```

Now, we need to train our classifier using gradient descent. We first define our training method and some variables for measuring our accuracy. The variable training will perform the gradient descent optimizer with a chosen LEARNING_RATE in order to try to minimize our loss function cross_entropy:

```
training =
tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(cross_e
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Now, we'll define a loop that repeats TRAIN_STEPS times; for each loop, it runs training, feeding in values from x_train and y_train using feed_dict. In order to calculate accuracy, it will run accuracy to classify the unseen data in x_test by comparing its y and y_test. It is vitally important that our test data was unseen and not used for training data. If a teacher were to give students a practice exam and use that same exam for the final exam, you would have a very biased measure of students' knowledge:

```
for i in range(TRAIN_STEPS+1):
sess.run(training, feed_dict={x: x_train, y_: y_train})
if i%100 == 0:
print('Training Step:' + str(i) + ' Accuracy = ' +
str(sess.run(accuracy, feed_dict={x: x_test, y_: y_test})) + ' > Loss
= ' + str(sess.run(cross_entropy, {x: x_train, y_: y_train})))
```

Don't forget to run your code and see your first neural network learning !

Fine, now that you ran your neural networkj maybe you will want to see the result ?

add those few lines to display a graph representing your network behavior

```
for i in range(10):
plt.subplot(2, 5, i+1)
weight = sess.run(W)[:,i]
plt.title(i)
plt.imshow(weight.reshape([28,28]), cmap=plt.get_cmap('seismic'))
frame1 = plt.gca()
frame1.axes.get_xaxis().set_visible(False)
frame1.axes.get_yaxis().set_visible(False)
plt.show()
```

Run your code again !

This is a visualization of our weights from 0-9. This is the most important aspect of our classifier. The bulk of the work of machine learning is figuring out what the optimal weights are; once they are calculated, you have the “cheat sheet” and can easily find answers. (This is part of why neural networks can be readily ported to mobile devices; the model, once trained, doesn’t take up that much room to store or computing power to calculate.) Our classifier makes its prediction by comparing how similar or different the digit is to the red and blue. I like to think the darker the red, the
better of a hit; white as neutral; and blue as misses.