Submitted by: Akshay Malhotra

# CSE 598: Assignment 1

## Handwritten Digit Recognition using TensorFlow and MNIST dataset

## Part 1: Applying Logistic Regression on MNIST handwritten digits.

1. Description of the model

   In this task, I implemented logistic regression to classify images and analyzed the performance of the model. Specifically, I worked on creating batches of the test data, developing the model, computing the loss and defining the optimizer.

   First, the dataset is imported with the support of the utils library. The train and test data are batched (batch size of 10000) and prepared for logistic regression. The weights and bias are initialized with the help of a normal distribution (mean=0, standard deviation=0.01) and zeros respectively. As there are 10 different, distinct classes and each image is of the dimension of (784,1) after flattening, the shape of the weight matrix is (784,10) and the shape of the bias matrix is (1,10).

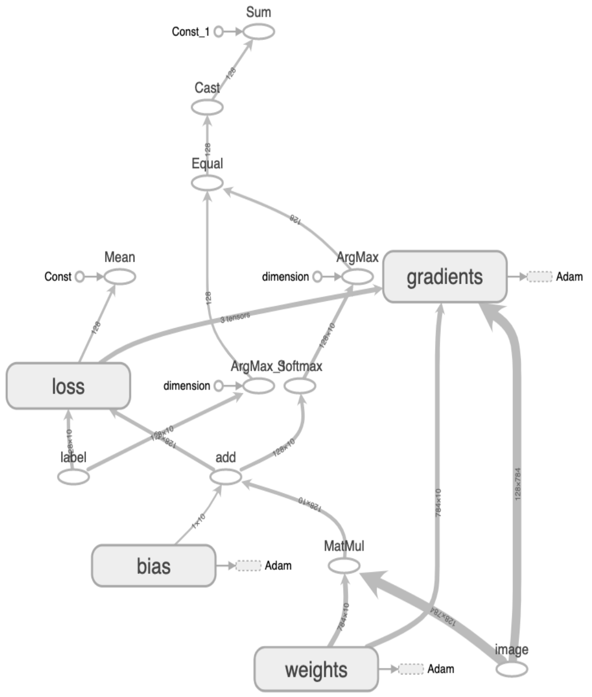
   Logistics regression is a linear model and can be represented as *Wx + b*. To compute this, I take advantage of the matrix multiplication function provided by TensorFlow. The output of this multiplication is then passed into a softmax logistic function to compute the probability of the image belonging to a certain class.

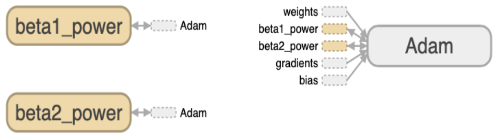
   To compute the loss, softmax cross-entropy is calculated between the logits and true labels and averaged out. In order to solve the optimization problem (obtain optimal values of weights and bias) such that loss in minimum, I used Adam Optimizer that is provided by Tensorflow. The learning rate is configured to 0.1.

2. Time Spent:
   In entirety, I spent **3 hours** on this part of the assignment.
   This includes understanding logistic regression and implementing the same.

Submitted by: Akshay Malhotra

## 3. Graph Representation





## 4. Accuracy and Analysis

Accuracy: 92.84%
Run Time: 12.8 seconds

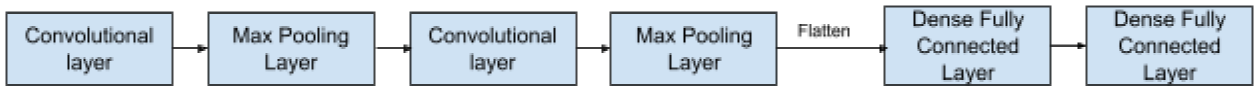
```
Epoch 1, Train Loss: 1.5635013580322266, Train Accuracy: 76.93636322021484, Test Accuracy: 77.63999938964844
Epoch 2, Train Loss: 0.8169341087341309, Train Accuracy: 86.86727142333984, Test Accuracy: 87.55000305175781
Epoch 3, Train Loss: 0.6427291035652161, Train Accuracy: 89.94727325439453, Test Accuracy: 89.96000671386719
Epoch 4, Train Loss: 0.5343993306159973, Train Accuracy: 90.44363403320312, Test Accuracy: 90.58000183105469
Epoch 5, Train Loss: 0.4639420211315155, Train Accuracy: 90.88909149169922, Test Accuracy: 90.88999938964844
Epoch 6, Train Loss: 0.41992509365081787, Train Accuracy: 91.52363586425781, Test Accuracy: 91.56999969482422
Epoch 7, Train Loss: 0.3740493059158325, Train Accuracy: 91.99091339111328, Test Accuracy: 91.98999786376953
Epoch 8, Train Loss: 0.34270942211151123, Train Accuracy: 92.40909576416016, Test Accuracy: 92.12999725341797
Epoch 9, Train Loss: 0.31568005681037903, Train Accuracy: 92.62181854248047, Test Accuracy: 92.23999786376953
Epoch 10, Train Loss: 0.29944586753845215, Train Accuracy: 92.79817962646484, Test Accuracy: 92.30999755859375
Epoch 11, Train Loss: 0.2830400764942169, Train Accuracy: 92.92909240722656, Test Accuracy: 92.52999877929688
Epoch 12, Train Loss: 0.26779571175575256, Train Accuracy: 93.14727020263672, Test Accuracy: 92.47999572753906
Epoch 13, Train Loss: 0.2602044641971588, Train Accuracy: 93.21636199951172, Test Accuracy: 92.66999816894531
Epoch 14, Train Loss: 0.25297990441322327, Train Accuracy: 93.33272552490234, Test Accuracy: 92.70999908447266
Epoch 15, Train Loss: 0.2499641329050064, Train Accuracy: 93.41999816894531, Test Accuracy: 92.73999786376953
Epoch 16, Train Loss: 0.24463017284870148, Train Accuracy: 93.3309097290039, Test Accuracy: 92.55999755859375
Epoch 17, Train Loss: 0.2445695847272873, Train Accuracy: 93.569091796875, Test Accuracy: 92.76000213623047
Epoch 18, Train Loss: 0.24123938381671906, Train Accuracy: 93.53091430664062, Test Accuracy: 92.91999816894531
Epoch 19, Train Loss: 0.23447759449481964, Train Accuracy: 93.57091522216797, Test Accuracy: 92.72000122070312
Epoch 20, Train Loss: 0.23519469797611237, Train Accuracy: 93.62000274658203, Test Accuracy: 92.68000030517578
Epoch 21, Train Loss: 0.23338757455348969, Train Accuracy: 93.4800033569336, Test Accuracy: 92.6500015258789
Epoch 22, Train Loss: 0.2334715723991394, Train Accuracy: 93.42363739013672, Test Accuracy: 92.51000213623047
Epoch 23, Train Loss: 0.23412300646305084, Train Accuracy: 93.62181854248047, Test Accuracy: 92.87999725341797
Epoch 24, Train Loss: 0.23148338496685028, Train Accuracy: 93.73091125488281, Test Accuracy: 92.88999938964844
Epoch 25, Train Loss: 0.22992946207523346, Train Accuracy: 93.69091033935547, Test Accuracy: 92.72000122070312
Epoch 26, Train Loss: 0.2271842211484909, Train Accuracy: 93.80363464355469, Test Accuracy: 92.77999877929688
Epoch 27, Train Loss: 0.2267737239599228, Train Accuracy: 93.79454040527344, Test Accuracy: 92.8499984741211
Epoch 28, Train Loss: 0.22578732669353485, Train Accuracy: 93.73635864257812, Test Accuracy: 92.79000091552734
Epoch 29, Train Loss: 0.22780121862888336, Train Accuracy: 93.20545196533203, Test Accuracy: 92.13999938964844
Epoch 30, Train Loss: 0.23538033664226532, Train Accuracy: 93.71454620361328, Test Accuracy: 92.83999633789062
Total Time 12.855441808700562 ms.
```

Submitted by: Akshay Malhotra

## Part 2: Applying Convolutional Neural Network on MNIST handwritten digits.

1. Description of the model

In the second part of the assignment, I take advantage of a convolutional neural network to classify images. The network that I implemented contains 2 convolutional layers, 2 pooling layers and 2 fully connected layers. Mentioned below is the sequential model that I implemented.



First, I import the dataset using tensorflow.keras and obtain the training and testing data with truth labels. After obtaining the data, I normalize it by dividing it by 255.0. This is because the image is represented by pixel intensities in the range of 0-255 and usually, normalization of data has yielded in a quicker learning time as gradients can be calculated quickly and each input variable has equal weightage. That being said, normalization of this dataset is not required as all pixel intensities values fall in the same range. After normalization, I reshape the data and prepare it for Tensorflow's API. The output labels are converted and represented in terms of one-hot encoding as our original output labels are categorical and not numerical.

Next, we prepare our convolutional neural network.

Layer 1: Convolutional Layer
A convolutional layer is used as the first layer that expects an input of the dimensions of (28,28,1). 32 filters are used in this layer and the dimension of each filter is (3,3). Stride is configured to 1, ie the filter moves in all directions by skipping 1 row or column, and padding is applied to the image. The output of this layer is passed through a rectified linear unit (ReLU) activation function.
This results in a convolved output that is of the same dimensions as the input but in the multiples of the number of filters (dimension of output is (28,28,32)).

Layer 2: Max Pooling Layer
Pooling is a technique of reducing the dimensionality of the data without losing out on the important features. This helps reduce the computation effort that would otherwise be invested in deriving a classification output. There are many types of pooling techniques available such as max pooling, min pooling and average pooling.
In this layer, I implement a max-pooling layer. The output of layer 1 is passed as input to this layer (dimension of (28,28,32)). The window size is configured to (2,2) and strides=2. As the window moves across the images, only the maximum value obtained by the window is maintained. As a result, the output obtained from this layer is half the size of the input. The output dimensions are (14,14,32).

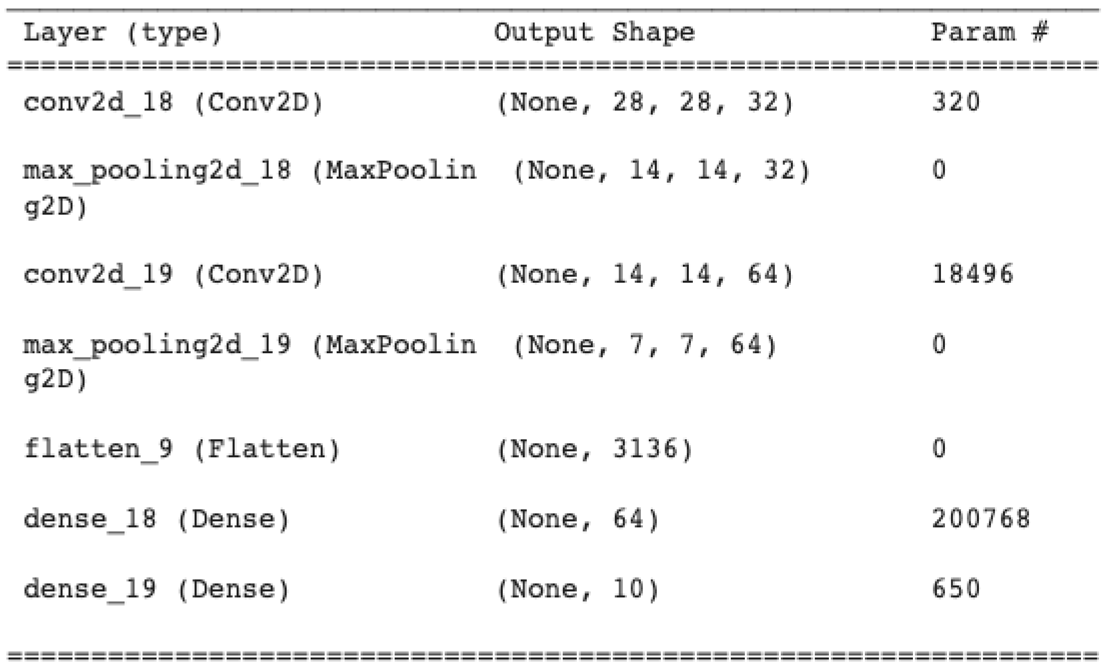
Layer 3: Convolutional Layer
A convolutional layer is implemented as the 3rd layer. 64 filters are used in this layer and the dimension of each filter is (3,3). Stride is configured to 1, ie the filter moves in all directions by skipping 1 row or column, and padding is applied to the image. The output of this layer is passed through a rectified linear unit (ReLU) activation function. The dimensions of the output obtained from this layer is (14,14,64).

Layer 4: Max Pooling Layer
I implement a max-pooling layer as the 4th layer. The window size is configured to (2,2) and strides=2. As a result, the output obtained from this layer is half the size of the input. The output dimensions are (7,7,64). I implement another layer that flattens this output to the dimension of (1,3136).

Layer 5: Dense Fully Connected Layer
As the 5th layer, I use a dense fully connected layer that consists of 64 neurons. The weights are multiplied with the outputs of the previous layer and a bias is added to obtain the result. This result is passed through a ReLU activation function before being passed to the next layer.

Layer 6: Dense Fully Connected Layer
This dense layer consists of only 10 neurons. The weights are multiplied with the outputs of the previous layer and a bias is added to obtain the result. This result is passed through a Softmax activation function to obtain class confidence results.

Mentioned below is a quick summary of the network.



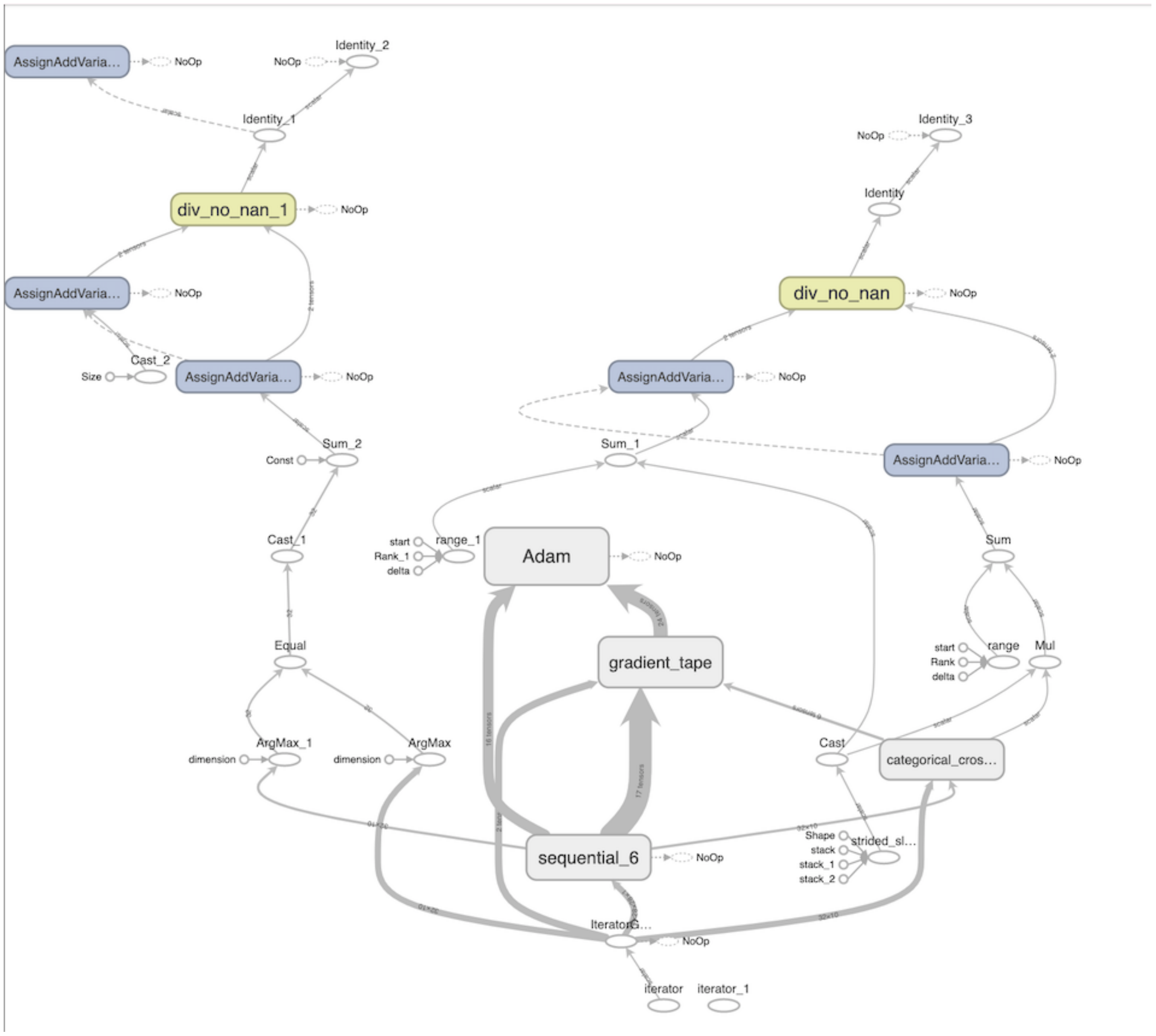
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_18 (Conv2D)          (None, 28, 28, 32)        320

 max_pooling2d_18 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_19 (Conv2D)          (None, 14, 14, 64)        18496

 max_pooling2d_19 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 flatten_9 (Flatten)         (None, 3136)              0

 dense_18 (Dense)            (None, 64)                200768

 dense_19 (Dense)            (None, 10)                650

=================================================================
```

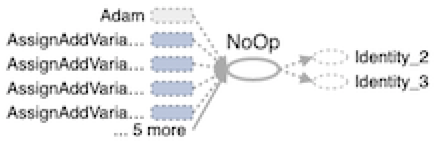
Once this network is prepared, it is compiled with a loss function (cross-entropy loss between the predicted labels and truth labels) and an Adam Optimizer to solve the optimization objective. The model is trained on the training data with 15 epochs and then evaluated to obtain performance metrics.

Submitted by: Akshay Malhotra

2. <u>Time Spent:</u>
   In entirety, I spent **10 hours** on this part of the assignment.
   This includes understanding the different layers and working of a CNN, understanding TensorFlow's API and implementing the network to obtain the results.

3. <u>Graph Representation</u>





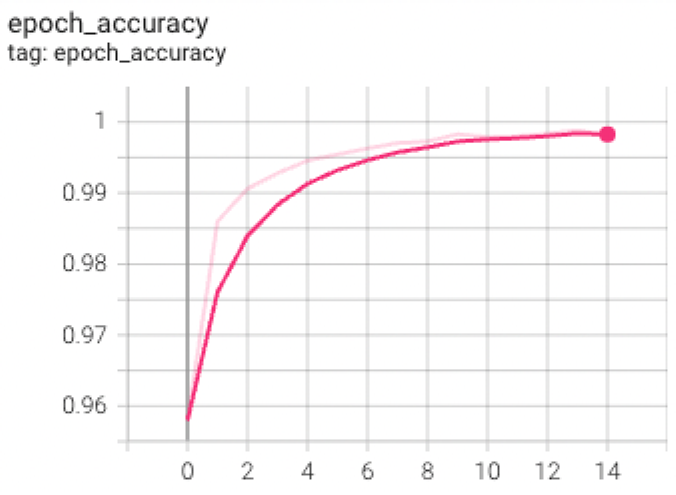
Submitted by: Akshay Malhotra

## 4. Accuracy and Analysis

Accuracy: 98.97%
Run Time: 15 epochs ~ 16 minutes

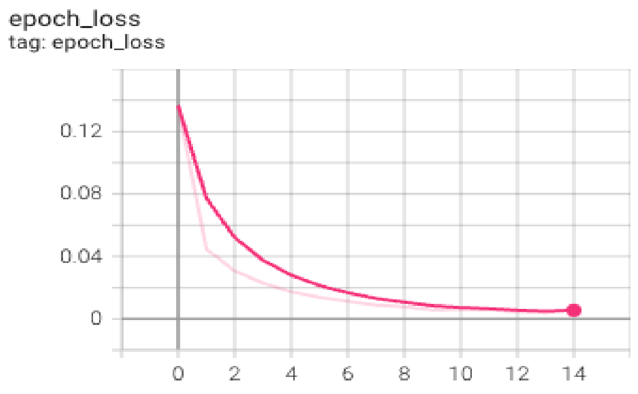
```
Epoch 1/15
1875/1875 [==============================] - 65s 34ms/step - loss: 0.1369 - accuracy: 0.9580
Epoch 2/15
1875/1875 [==============================] - 65s 34ms/step - loss: 0.0448 - accuracy: 0.9860
Epoch 3/15
1875/1875 [==============================] - 65s 34ms/step - loss: 0.0307 - accuracy: 0.9906
Epoch 4/15
1875/1875 [==============================] - 65s 35ms/step - loss: 0.0230 - accuracy: 0.9928
Epoch 5/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0174 - accuracy: 0.9945
Epoch 6/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0138 - accuracy: 0.9954
Epoch 7/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0112 - accuracy: 0.9963
Epoch 8/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0089 - accuracy: 0.9970
Epoch 9/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0077 - accuracy: 0.9973
Epoch 10/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0057 - accuracy: 0.9983
Epoch 11/15
1875/1875 [==============================] - 63s 34ms/step - loss: 0.0059 - accuracy: 0.9979
Epoch 12/15
1875/1875 [==============================] - 63s 34ms/step - loss: 0.0058 - accuracy: 0.9980
Epoch 13/15
1875/1875 [==============================] - 63s 34ms/step - loss: 0.0043 - accuracy: 0.9984
Epoch 14/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0040 - accuracy: 0.9988
Epoch 15/15
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0063 - accuracy: 0.9981
313/313 [==============================] - 3s 10ms/step - loss: 0.0523 - accuracy: 0.9897
```

*Results of running CNN ~ 15 epochs*



*CNN: Accuracy vs Epoch*

Submitted by: Akshay Malhotra



*CNN: Epoch Loss vs Epoch*

## Problems encountered during assignment:

I faced a few issues while working on this assignment, however, they did not hinder me from completing my assignment. Mentioned below are a few issues that I faced.

- Installing Tensorflow on M1 MacBook
  A base version of Tensorflow is available on major distribution channels that work on M1 ARM processors, however, the latest version of Tensorflow is currently not being distributed on popular channels. The installation process was tedious in order to get Tensorboard to work.

- Understanding differences between Tensorflow V1 and Tensorflow V2
  In certain parts of the assignment, I had to use Tensorflow V1 API to get to work as they did not seem to work on Tensorflow V2. Understanding the difference between V1 and V2 introduced a learning curve.

- Performance of different optimizers
  I experimented with a few optimizers in my CNN to understand how they affect the performance.

| **Optimizer** | **Epoch** | **Learning Rate** | **Accuracy** |
|---|---|---|---|
| Adam | 5 | 0.001 (default) | 98.9% |
| Adadelta | 5 | 0.001 (default) | 73.6% |
| Adagrad | 5 | 0.001 (default) | 93.8% |
| SGD | 5 | 0.01 (default) | 97.3% |

## Total Time spent on Assignment

Part 1 Estimate - 3 hours
Part 2 Estimate - 10 hours
Total Time Taken - 13 hours