Artificial Neural Networks

Assignment 2

Team members:

Arvind Narayan Srinivasan(105187866)

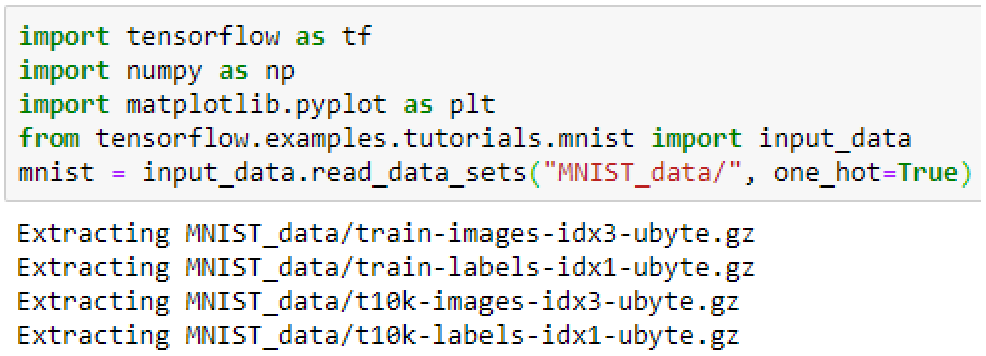
Gundeep Singh(110008447)

Vigneshwaran Balasubramani(105221382)

Question 1

Download the benchmark dataset MNIST from http://yann.lecun.com/ exdb/mnist/. Implement multiclass logistic regression and try it on MNIST.

We imported the MNIST dataset from tensorflow library as below



```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Fig1: *MNIST dataset imported from Tensorflow*

Train and Test dataset were split and prepared for the computation and SGD is used on the training data as an optimizer as shown in *Fig2*

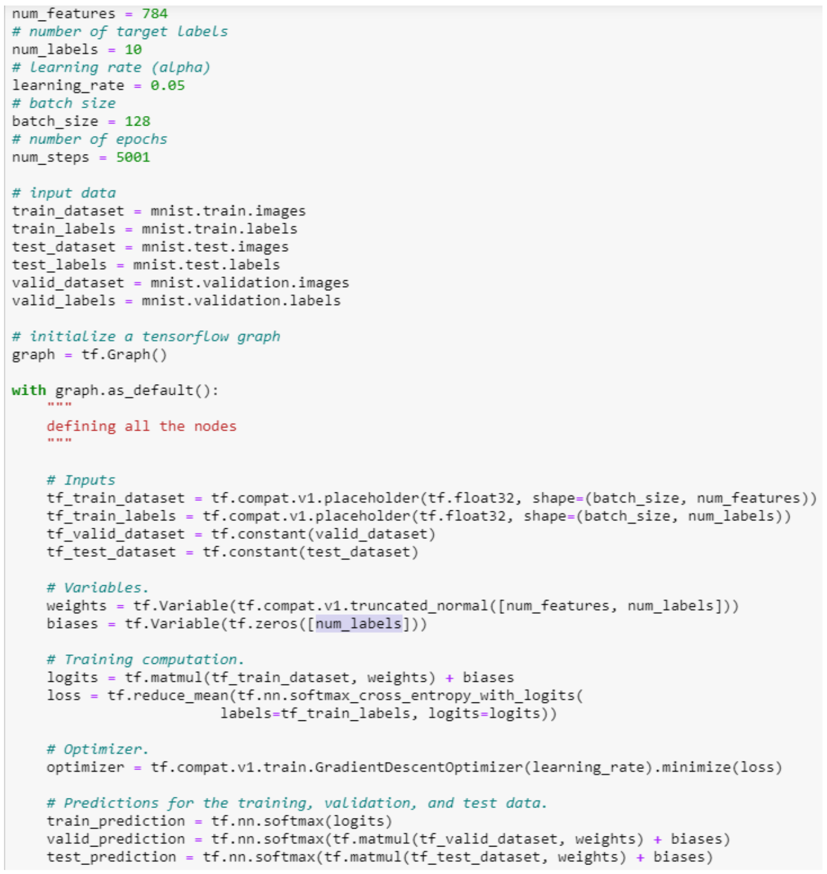
The weight matrix is computing using random values from a normal distribution and the biases are initialized to zero

Softmax and cross-entropy are computed on all the training examples and their average is also taken using reduce mean function in tensorflow

We use gradient descent optimizer to reduce the loss in the training examples and we also print them during the computation phase in *Fig3*

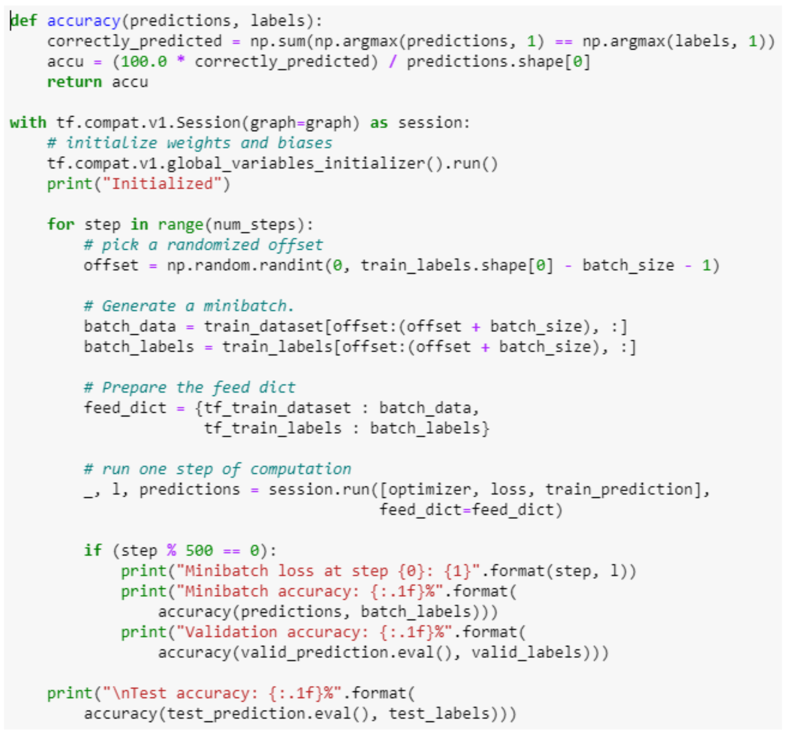
A tensor flow session is created and computations are run using a step function to display the output as in *Fig4*

We can see the increase in accuracy of each step from validation accuracy starting at 14.3% to the total test accuracy of 86.5%



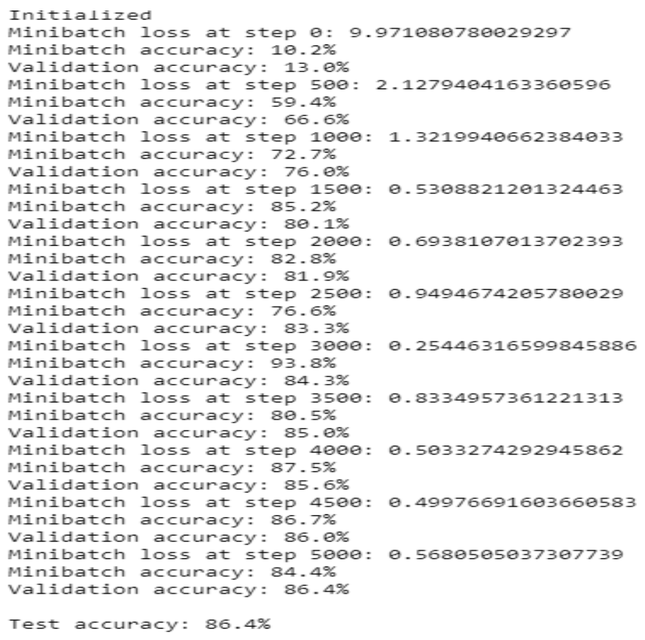
```python
num_features = 784
# number of target labels
num_labels = 10
# learning rate (alpha)
learning_rate = 0.05
# batch size
batch_size = 128
# number of epochs
num_steps = 5001

# input data
train_dataset = mnist.train.images
train_labels = mnist.train.labels
test_dataset = mnist.test.images
test_labels = mnist.test.labels
valid_dataset = mnist.validation.images
valid_labels = mnist.validation.labels

# initialize a tensorflow graph
graph = tf.Graph()

with graph.as_default():
    """
    defining all the nodes
    """

    # Inputs
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_features))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(valid_dataset)
    tf_test_dataset = tf.constant(test_dataset)

    # Variables.
    weights = tf.Variable(tf.compat.v1.truncated_normal([num_features, num_labels]))
    biases = tf.Variable(tf.zeros([num_labels]))

    # Training computation.
    logits = tf.matmul(tf_train_dataset, weights) + biases
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
                        labels=tf_train_labels, logits=logits))

    # Optimizer.
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(loss)

    # Predictions for the training, validation, and test data.
    train_prediction = tf.nn.softmax(logits)
    valid_prediction = tf.nn.softmax(tf.matmul(tf_valid_dataset, weights) + biases)
    test_prediction = tf.nn.softmax(tf.matmul(tf_test_dataset, weights) + biases)
```

Fig2: *Preparing data for running multiclass logistic regression*



```python
def accuracy(predictions, labels):
    correctly_predicted = np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1))
    accu = (100.0 * correctly_predicted) / predictions.shape[0]
    return accu

with tf.compat.v1.Session(graph=graph) as session:
    # initialize weights and biases
    tf.compat.v1.global_variables_initializer().run()
    print("Initialized")

    for step in range(num_steps):
        # pick a randomized offset
        offset = np.random.randint(0, train_labels.shape[0] - batch_size - 1)

        # Generate a minibatch.
        batch_data = train_dataset[offset:(offset + batch_size), :]
        batch_labels = train_labels[offset:(offset + batch_size), :]

        # Prepare the feed dict
        feed_dict = {tf_train_dataset : batch_data,
                     tf_train_labels : batch_labels}

        # run one step of computation
        _, l, predictions = session.run([optimizer, loss, train_prediction],
                                        feed_dict=feed_dict)

        if (step % 500 == 0):
            print("Minibatch loss at step {0}: {1}".format(step, l))
            print("Minibatch accuracy: {:.1f}%".format(
                accuracy(predictions, batch_labels)))
            print("Validation accuracy: {:.1f}%".format(
                accuracy(valid_prediction.eval(), valid_labels)))

    print("\nTest accuracy: {:.1f}%".format(
        accuracy(test_prediction.eval(), test_labels)))
```

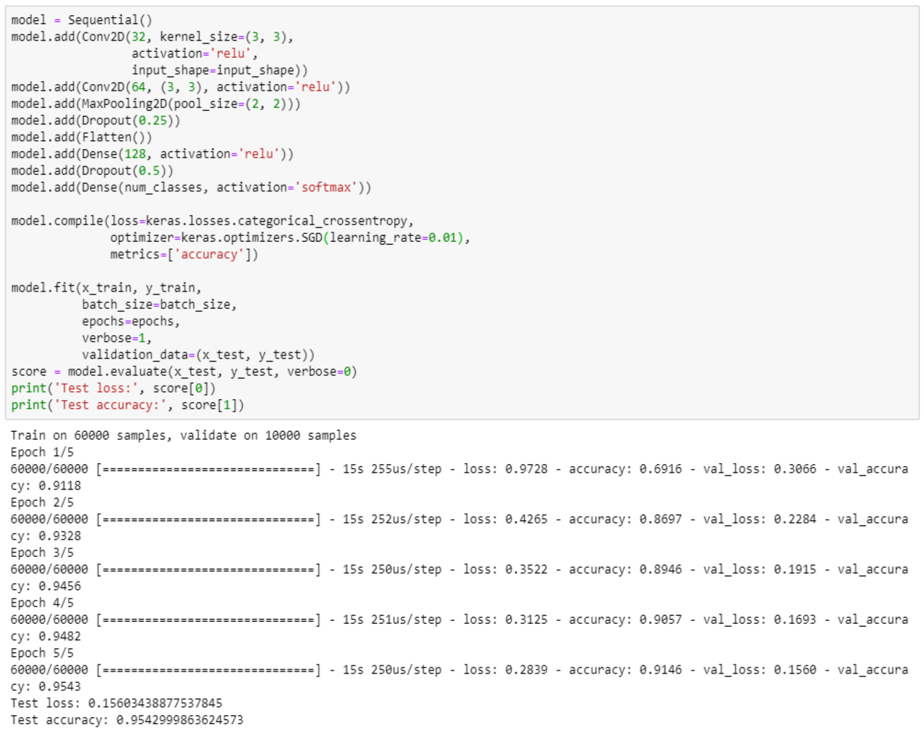
Fig3: *Running Multiclass logistic regression*



```
Initialized
Minibatch loss at step 0: 9.971080780029297
Minibatch accuracy: 10.2%
Validation accuracy: 13.0%
Minibatch loss at step 500: 2.1279404163360596
Minibatch accuracy: 59.4%
Validation accuracy: 66.6%
Minibatch loss at step 1000: 1.3219940662384033
Minibatch accuracy: 72.7%
Validation accuracy: 76.0%
Minibatch loss at step 1500: 0.5308821201324463
Minibatch accuracy: 85.2%
Validation accuracy: 80.1%
Minibatch loss at step 2000: 0.6938107013702393
Minibatch accuracy: 82.8%
Validation accuracy: 81.9%
Minibatch loss at step 2500: 0.9494674205780029
Minibatch accuracy: 76.6%
Validation accuracy: 83.3%
Minibatch loss at step 3000: 0.25446316599845886
Minibatch accuracy: 93.8%
Validation accuracy: 84.3%
Minibatch loss at step 3500: 0.8334957361221313
Minibatch accuracy: 80.5%
Validation accuracy: 85.0%
Minibatch loss at step 4000: 0.5033274292945862
Minibatch accuracy: 87.5%
Validation accuracy: 85.6%
Minibatch loss at step 4500: 0.49976691603660583
Minibatch accuracy: 86.7%
Validation accuracy: 86.0%
Minibatch loss at step 5000: 0.5680505037307739
Minibatch accuracy: 84.4%
Validation accuracy: 86.4%

Test accuracy: 86.4%
```

Fig4: *Output from Multiclass logistic regression*

1. Try the basic minibatch SGD. It is recommended to try different initializations, different batch sizes, and different learning rates, in order to get a sense about how to tune the hyperparameters (batch size, and, learning rate). Remember to create and use validation dataset!



```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(learning_rate=0.01),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 15s 255us/step - loss: 0.9728 - accuracy: 0.6916 - val_loss: 0.3066 - val_accura
cy: 0.9118
Epoch 2/5
60000/60000 [==============================] - 15s 252us/step - loss: 0.4265 - accuracy: 0.8697 - val_loss: 0.2284 - val_accura
cy: 0.9328
Epoch 3/5
60000/60000 [==============================] - 15s 250us/step - loss: 0.3522 - accuracy: 0.8946 - val_loss: 0.1915 - val_accura
cy: 0.9456
Epoch 4/5
60000/60000 [==============================] - 15s 251us/step - loss: 0.3125 - accuracy: 0.9057 - val_loss: 0.1693 - val_accura
cy: 0.9482
Epoch 5/5
60000/60000 [==============================] - 15s 250us/step - loss: 0.2839 - accuracy: 0.9146 - val_loss: 0.1560 - val_accura
cy: 0.9543
Test loss: 0.15603438877537845
Test accuracy: 0.9542999863624573
```

Fig5: *Basic Minibatch SGD using Keras MNIST dataset with LR = 0.01 and Batch size = 128*

We used the MNIST dataset from Keras and the SGD fucntion from Keras Optimizers to implement a basic minibatch SGD. When used with the standard learning rate of **0.01** and batch size of **128** we got the test accuracy to be **0.95** (For 5 Epochs).

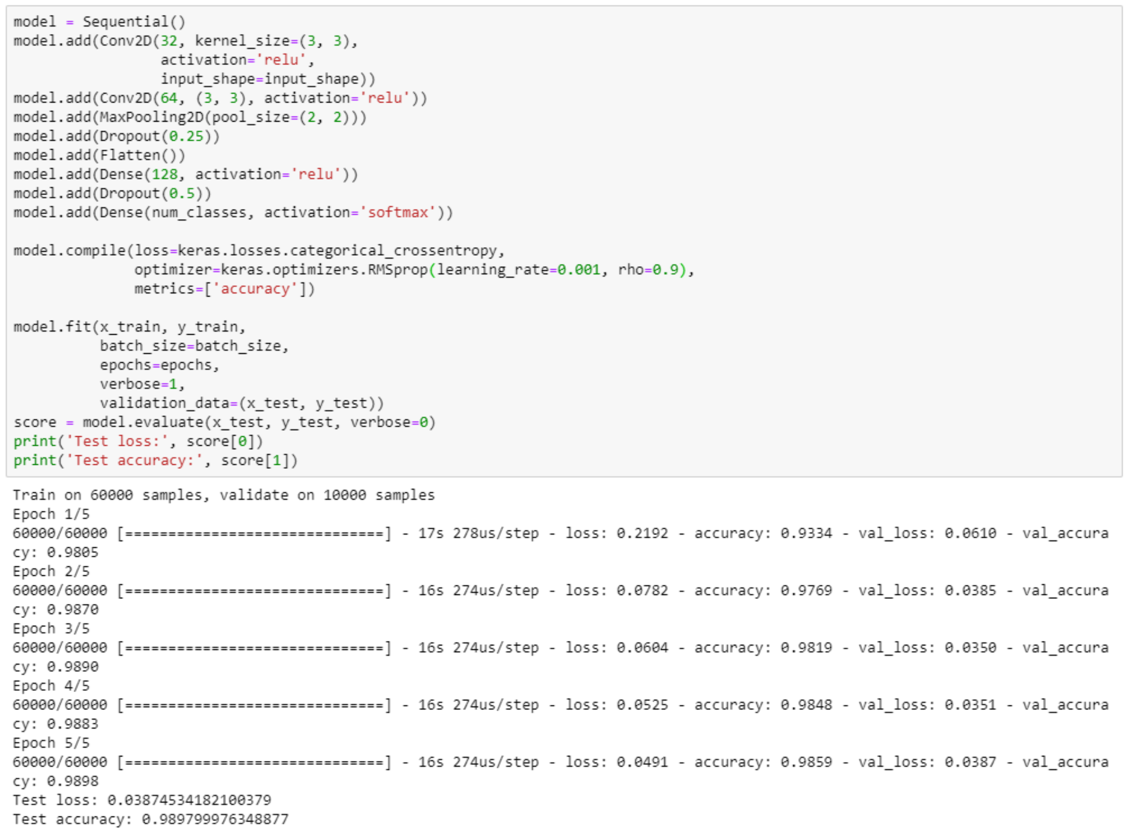
We tried the same algorithm with learning rate **0.1** and batch size **150** with number of classes of **20** we got the test accuracy to be **0.98** (For 10 Epochs).

For the batch size **200** and learning rate of **0.5** we got the test accuracy as 0.1 (For 10 Epochs).

As we can observe from the above statistics we can see that increasing the learning rate and batch size looks like a case of overfitting. Hence the hyperparameters should be kept at an ideal values of **0.01** for learning rate and close to **130** for batch size.

2. It is recommended to try, at least, another optimization method (SGD momentum, RMSProp, RMSProp momentum, AdaDelta, or Adam) and compare its performances to those of the basic minibatch SGD on the MNIST dataset.

We saw the standard minibatch SGDs performance from *Fig5*. Lets compare the same with RMSProp, AdaDelta and Adam. We used the Keras optimizers function which comprises the functions for above.

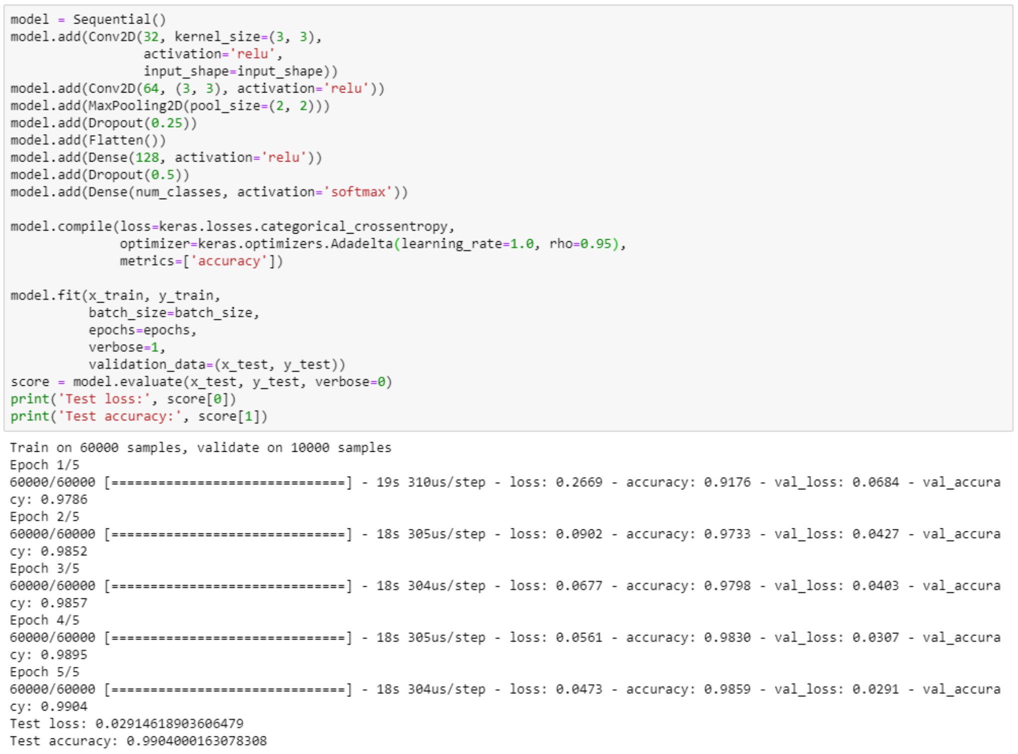


```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 17s 278us/step - loss: 0.2192 - accuracy: 0.9334 - val_loss: 0.0610 - val_accura
cy: 0.9805
Epoch 2/5
60000/60000 [==============================] - 16s 274us/step - loss: 0.0782 - accuracy: 0.9769 - val_loss: 0.0385 - val_accura
cy: 0.9870
Epoch 3/5
60000/60000 [==============================] - 16s 274us/step - loss: 0.0604 - accuracy: 0.9819 - val_loss: 0.0350 - val_accura
cy: 0.9890
Epoch 4/5
60000/60000 [==============================] - 16s 274us/step - loss: 0.0525 - accuracy: 0.9848 - val_loss: 0.0351 - val_accura
cy: 0.9883
Epoch 5/5
60000/60000 [==============================] - 16s 274us/step - loss: 0.0491 - accuracy: 0.9859 - val_loss: 0.0387 - val_accura
cy: 0.9898
Test loss: 0.03874534182100379
Test accuracy: 0.989799976348877
```

Fig6: *RMSprop using Keras MNIST dataset with LR = 0.001 and Batch size = 128*

We can see that the test accuracy is **0.98** with the learning rate **0.001**.

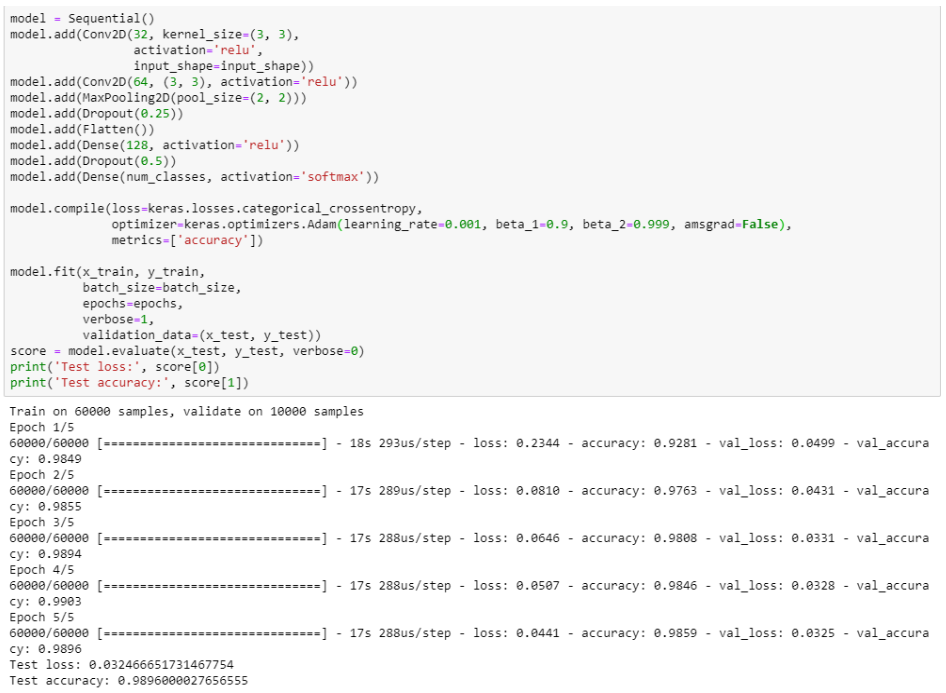


```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(learning_rate=1.0, rho=0.95),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 19s 310us/step - loss: 0.2669 - accuracy: 0.9176 - val_loss: 0.0684 - val_accura
cy: 0.9786
Epoch 2/5
60000/60000 [==============================] - 18s 305us/step - loss: 0.0902 - accuracy: 0.9733 - val_loss: 0.0427 - val_accura
cy: 0.9852
Epoch 3/5
60000/60000 [==============================] - 18s 304us/step - loss: 0.0677 - accuracy: 0.9798 - val_loss: 0.0403 - val_accura
cy: 0.9857
Epoch 4/5
60000/60000 [==============================] - 18s 305us/step - loss: 0.0561 - accuracy: 0.9830 - val_loss: 0.0307 - val_accura
cy: 0.9895
Epoch 5/5
60000/60000 [==============================] - 18s 304us/step - loss: 0.0473 - accuracy: 0.9859 - val_loss: 0.0291 - val_accura
cy: 0.9904
Test loss: 0.02914618903606479
Test accuracy: 0.9904000163078308
```

Fig7: *Adadelta using Keras MNIST dataset with LR = 1.0 and Batch size = 128*

The test accuracy when we used Adadelta is **0.99** with learning rate **1.0**.



```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 18s 293us/step - loss: 0.2344 - accuracy: 0.9281 - val_loss: 0.0499 - val_accura
cy: 0.9849
Epoch 2/5
60000/60000 [==============================] - 17s 289us/step - loss: 0.0810 - accuracy: 0.9763 - val_loss: 0.0431 - val_accura
cy: 0.9855
Epoch 3/5
60000/60000 [==============================] - 17s 288us/step - loss: 0.0646 - accuracy: 0.9808 - val_loss: 0.0331 - val_accura
cy: 0.9894
Epoch 4/5
60000/60000 [==============================] - 17s 288us/step - loss: 0.0507 - accuracy: 0.9846 - val_loss: 0.0328 - val_accura
cy: 0.9903
Epoch 5/5
60000/60000 [==============================] - 17s 288us/step - loss: 0.0441 - accuracy: 0.9859 - val_loss: 0.0325 - val_accura
cy: 0.9896
Test loss: 0.032466651731467754
Test accuracy: 0.9896000027656555
```

Fig8: *Adam using Keras MNIST dataset with LR = 0.001 and Batch size = 128*

The test accuracy when we used Adam is **0.989** with a learning rate **0.001**.

When we compare the standard performances of RMSprop, Adadelta and Adam with Minibatch SGD we can see that the former methods prove to give more accuracy than standard Minibatch SGD.

Question #2 Consider the $L2$-regularized multiclass logistic regression. That is, add to the logistic regression loss a regularization term that represents the L2 norm of the parameters. More precisely, the regularization term is

$$\lambda\Sigma_i(\|w_i\|2 + \|b_i\|2)$$

where $\{wi, bi\}$ are all the parameters in the logistic regression, and $\lambda \in R$ is the regularization hyper-parameter. Typically, λ is about C/n where n is the number of data points and C is some constant in [0.01,100] (need to tune C). Run the regularized multiclass logistic regression on MNIST, using the basic minibatch SGD, and compare its results to those of the basic minibatch SGD with non-regularized loss, in Question #1.

Solution:

In the first question, we had implemented multiclass logistic regression on MNIST with minibatch SGD and compared it with other optimization methods. We use the same minibatch SGD but with regularized input with L2 norm.

In Keras, we implement the regularizers to apply penalties layer wise and avoid overfitting by preventing high coefficients. The regularizers are embedded inside the loss function itself and there are primarily two API regularizers added to the input layer: the kernel and the bias regularizer.

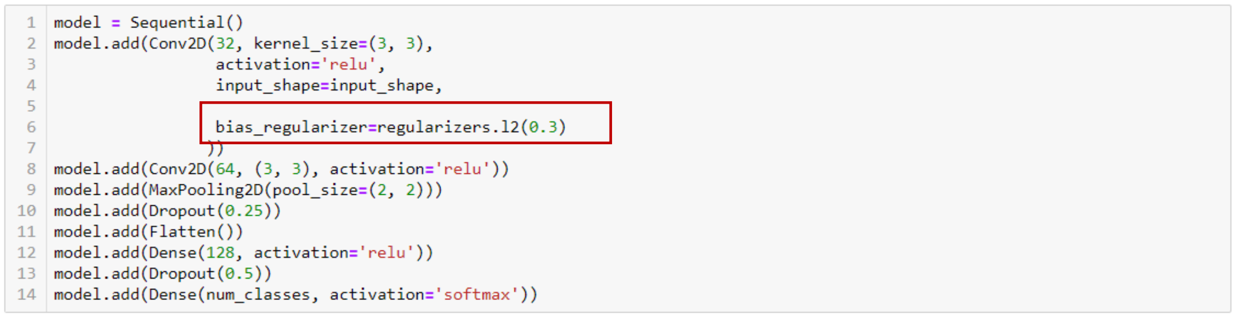
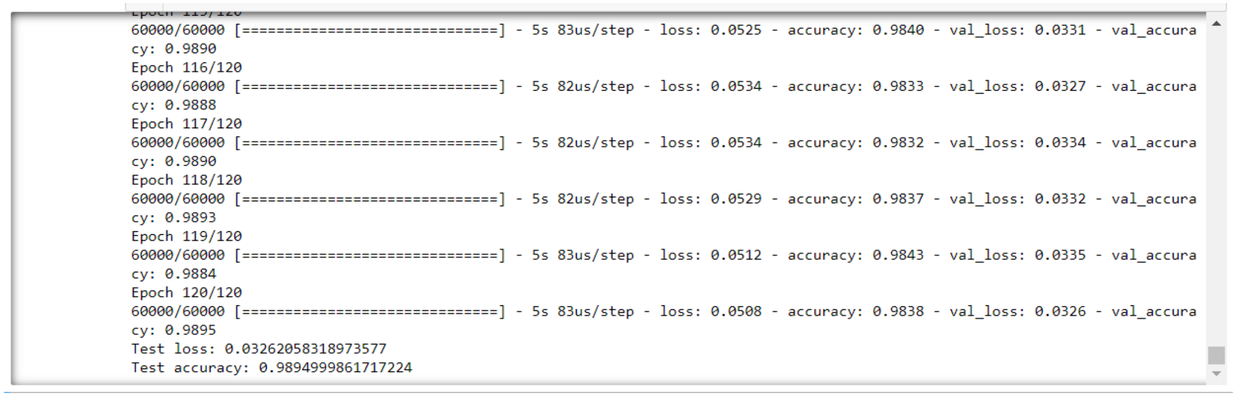

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape,

                 bias_regularizer=regularizers.l2(0.3)
                ))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

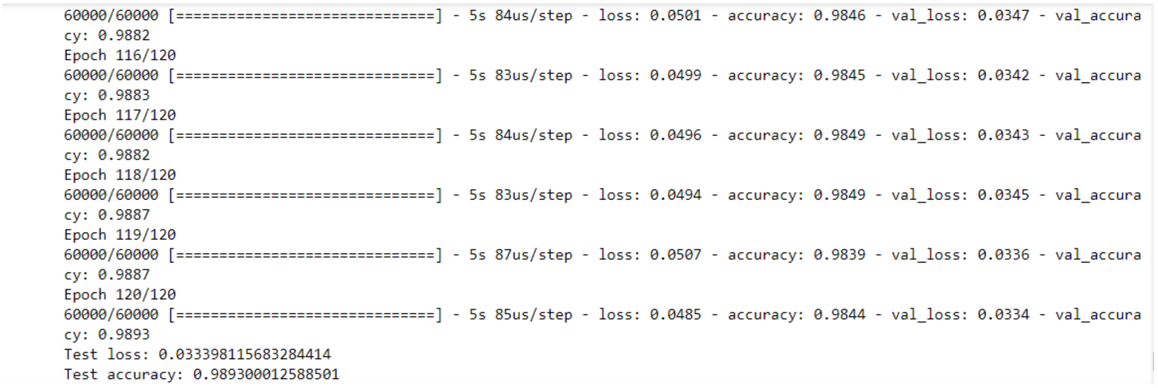
***Fig9**: Neural Network setup with L2 regularizer*

We have added the regularizers to the input layer with value of C=0.3.



```
60000/60000 [==============================] - 5s 83us/step - loss: 0.0525 - accuracy: 0.9840 - val_loss: 0.0331 - val_accura
cy: 0.9890
Epoch 116/120
60000/60000 [==============================] - 5s 82us/step - loss: 0.0534 - accuracy: 0.9833 - val_loss: 0.0327 - val_accura
cy: 0.9888
Epoch 117/120
60000/60000 [==============================] - 5s 82us/step - loss: 0.0534 - accuracy: 0.9832 - val_loss: 0.0334 - val_accura
cy: 0.9890
Epoch 118/120
60000/60000 [==============================] - 5s 82us/step - loss: 0.0529 - accuracy: 0.9837 - val_loss: 0.0332 - val_accura
cy: 0.9893
Epoch 119/120
60000/60000 [==============================] - 5s 83us/step - loss: 0.0512 - accuracy: 0.9843 - val_loss: 0.0335 - val_accura
cy: 0.9884
Epoch 120/120
60000/60000 [==============================] - 5s 83us/step - loss: 0.0508 - accuracy: 0.9838 - val_loss: 0.0326 - val_accura
cy: 0.9895
Test loss: 0.03262058318973577
Test accuracy: 0.9894999861717224
```

Fig10: *Test Accuracy with L2 norm*



```
60000/60000 [==============================] - 5s 84us/step - loss: 0.0501 - accuracy: 0.9846 - val_loss: 0.0347 - val_accura
cy: 0.9882
Epoch 116/120
60000/60000 [==============================] - 5s 83us/step - loss: 0.0499 - accuracy: 0.9845 - val_loss: 0.0342 - val_accura
cy: 0.9883
Epoch 117/120
60000/60000 [==============================] - 5s 84us/step - loss: 0.0496 - accuracy: 0.9849 - val_loss: 0.0343 - val_accura
cy: 0.9882
Epoch 118/120
60000/60000 [==============================] - 5s 83us/step - loss: 0.0494 - accuracy: 0.9849 - val_loss: 0.0345 - val_accura
cy: 0.9887
Epoch 119/120
60000/60000 [==============================] - 5s 87us/step - loss: 0.0507 - accuracy: 0.9839 - val_loss: 0.0336 - val_accura
cy: 0.9887
Epoch 120/120
60000/60000 [==============================] - 5s 85us/step - loss: 0.0485 - accuracy: 0.9844 - val_loss: 0.0334 - val_accura
cy: 0.9893
Test loss: 0.033398115683284414
Test accuracy: 0.989300012588501
```

Fig11: *Test Accuracy without L2 norm*

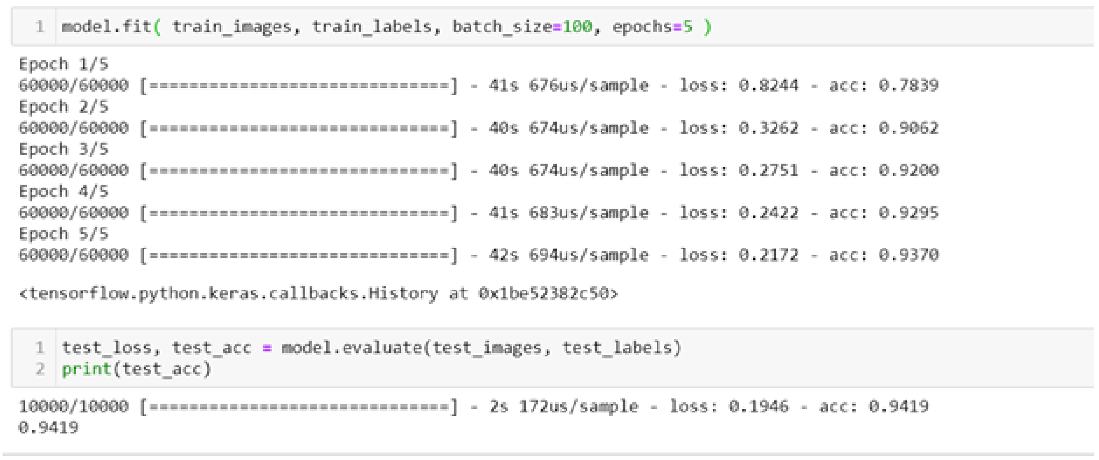
From the test accuracies of testing the MNIST data with and without regularization, we can infer that there is a slight increase in the accuracy of the test data. This is due to an added parameter that prevents the model from overfitting.

Question #3 Going above and beyond Question-1 and Question-2, investigate the basic mini batch SGD with, at least, another regularization method discussed in class (L 1 , data augmentation, noise robustness, early stopping, sparse representation, bagging, or dropout). Currently, L 2 norm, early stopping, and dropout are the most frequently used regularization methods. You may need to read Chapter-7, which I have started to cover in class. You may even try CNN if time allows you.

Solution:

In this problem we try different regularization techniques to compare out the variations in the outputs of all the cases.

Given below is the output that we get without applying any regularization . A sequential model is used with Convolutional 2D layer and '*relu*' as the activation function. However for the final output layer, softmax is used as the activation function as the nature of the problem is multi class classification. Stochastic Gradient descent is used as the optimizer with batch size='*100*' and learning rate as '*0.01*'.



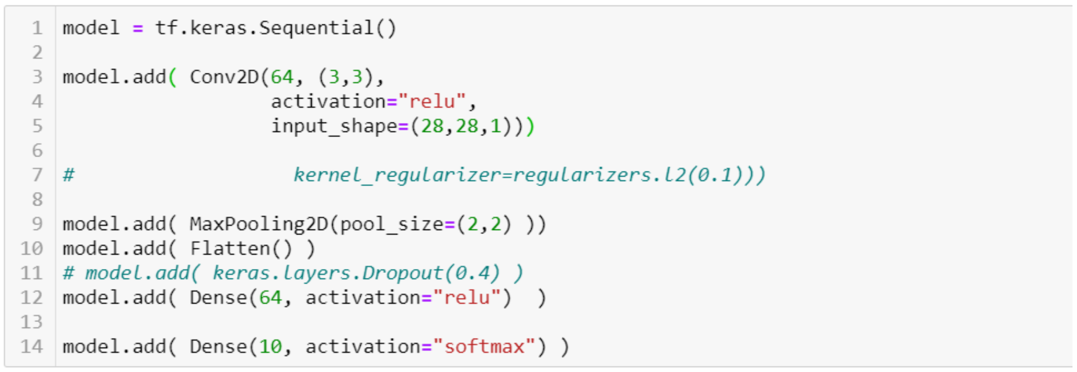
```
model.fit( train_images, train_labels, batch_size=100, epochs=5 )
```

```
Epoch 1/5
60000/60000 [==============================] - 41s 676us/sample - loss: 0.8244 - acc: 0.7839
Epoch 2/5
60000/60000 [==============================] - 40s 674us/sample - loss: 0.3262 - acc: 0.9062
Epoch 3/5
60000/60000 [==============================] - 40s 674us/sample - loss: 0.2751 - acc: 0.9200
Epoch 4/5
60000/60000 [==============================] - 41s 683us/sample - loss: 0.2422 - acc: 0.9295
Epoch 5/5
60000/60000 [==============================] - 42s 694us/sample - loss: 0.2172 - acc: 0.9370

<tensorflow.python.keras.callbacks.History at 0x1be52382c50>
```

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(test_acc)
```

```
10000/10000 [==============================] - 2s 172us/sample - loss: 0.1946 - acc: 0.9419
0.9419
```

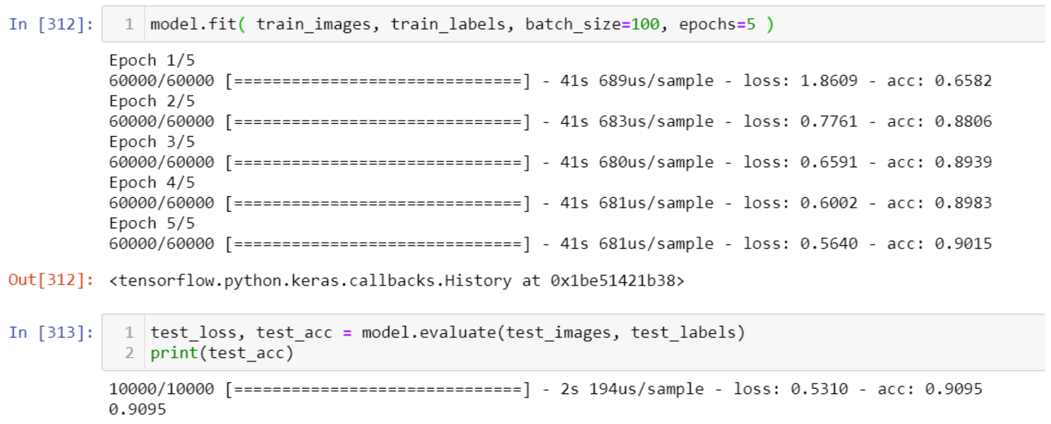
Fig12: *Output obtained without any regularization.*



```
model = tf.keras.Sequential()

model.add( Conv2D(64, (3,3),
                  activation="relu",
                  input_shape=(28,28,1)))

#                   kernel_regularizer=regularizers.l2(0.1)))

model.add( MaxPooling2D(pool_size=(2,2) ))
model.add( Flatten() )
# model.add( keras.layers.Dropout(0.4) )
model.add( Dense(64, activation="relu")  )

model.add( Dense(10, activation="softmax") )
```

Fig13: *Learning Model*

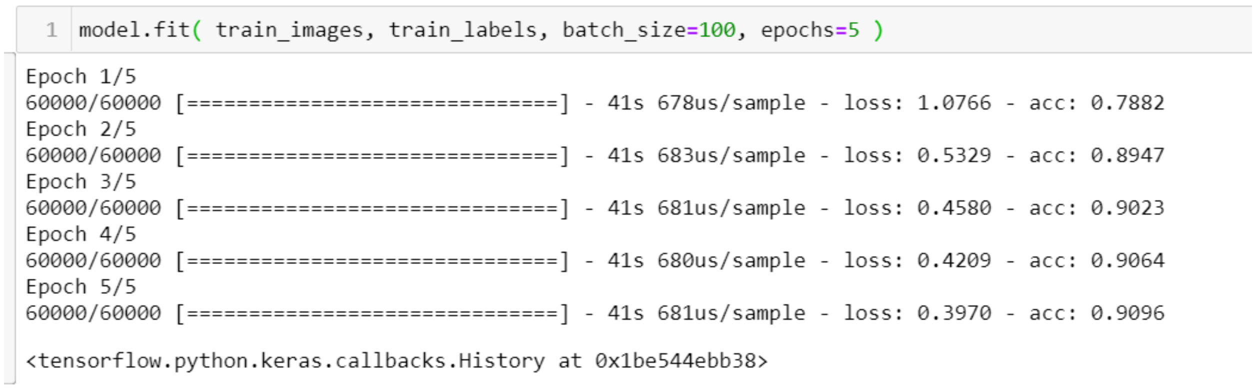
L1 Regularization: The first regularization technique used is the L1 kernel regularization with lambda(c) equal to ‘0.1’.Below picture shows the output obtained using the l1 regularization. Its observed that the training efficiency falls to a great extent but the test accuracy is affected as well making it not suitable over the regularization techniques shown further in the report:



```
In [312]:  model.fit( train_images, train_labels, batch_size=100, epochs=5 )

            Epoch 1/5
            60000/60000 [==============================] - 41s 689us/sample - loss: 1.8609 - acc: 0.6582
            Epoch 2/5
            60000/60000 [==============================] - 41s 683us/sample - loss: 0.7761 - acc: 0.8806
            Epoch 3/5
            60000/60000 [==============================] - 41s 680us/sample - loss: 0.6591 - acc: 0.8939
            Epoch 4/5
            60000/60000 [==============================] - 41s 681us/sample - loss: 0.6002 - acc: 0.8983
            Epoch 5/5
            60000/60000 [==============================] - 41s 681us/sample - loss: 0.5640 - acc: 0.9015

Out[312]:   <tensorflow.python.keras.callbacks.History at 0x1be51421b38>

In [313]:   test_loss, test_acc = model.evaluate(test_images, test_labels)
            print(test_acc)

            10000/10000 [==============================] - 2s 194us/sample - loss: 0.5310 - acc: 0.9095
            0.9095
```

Fig14: *L1 Regularization with c=0.1*

L2 Regularization: The second regularization technique thus used is the L2 regularization with lambda=’*0.1*’.
The below figure demonstrates the output thus achieved and showcases a decline in the training set of the original data set. The testing efficiency is better than that of L1 but does not improvise sufficiently the original condition where no regularization technique is used.



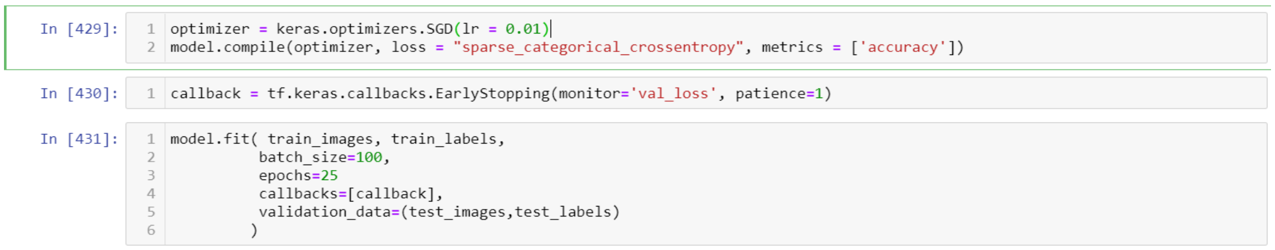
```
model.fit( train_images, train_labels, batch_size=100, epochs=5 )

Epoch 1/5
60000/60000 [==============================] - 41s 678us/sample - loss: 1.0766 - acc: 0.7882
Epoch 2/5
60000/60000 [==============================] - 41s 683us/sample - loss: 0.5329 - acc: 0.8947
Epoch 3/5
60000/60000 [==============================] - 41s 681us/sample - loss: 0.4580 - acc: 0.9023
Epoch 4/5
60000/60000 [==============================] - 41s 680us/sample - loss: 0.4209 - acc: 0.9064
Epoch 5/5
60000/60000 [==============================] - 41s 681us/sample - loss: 0.3970 - acc: 0.9096

<tensorflow.python.keras.callbacks.History at 0x1be544ebb38>
```

Fig15: *L2 Regularization with c=0.1*

The test set efficiency however is approximately 92% .

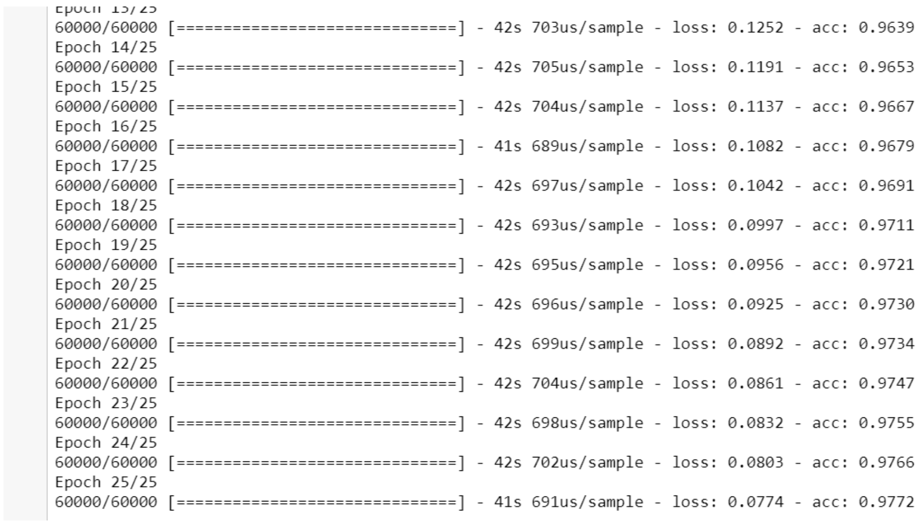
Early Stopping: We use yet another regularization technique called the early stopping in which a validation set is provided to compare the results and stop the training at a point where the further steps are redundant and doesn't improve the model any further.

The figures below show the implementation of Early Stopping and the outputs thus observed. The model was run again for this specific regularization test and the number of epochs were increased to 25.



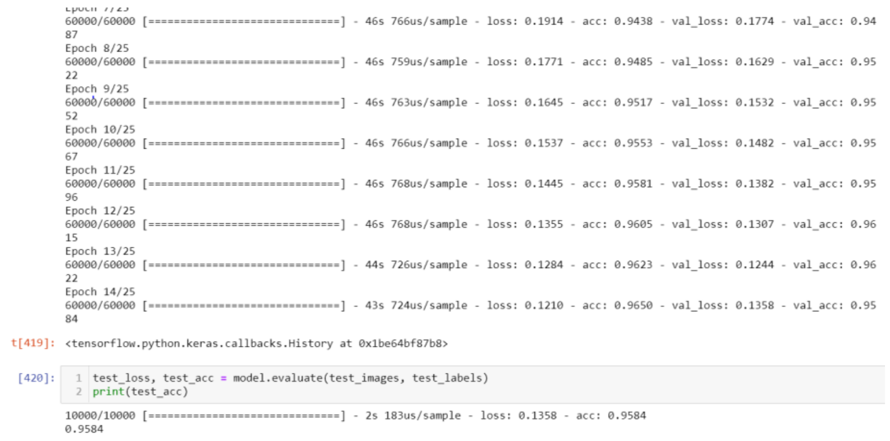
```
In [429]: optimizer = keras.optimizers.SGD(lr = 0.01)
          model.compile(optimizer, loss = "sparse_categorical_crossentropy", metrics = ['accuracy'])

In [430]: callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=1)

In [431]: model.fit( train_images, train_labels,
                    batch_size=100,
                    epochs=25
                    callbacks=[callback],
                    validation_data=(test_images,test_labels)
                   )
```

Fig16: *Early Stopping with patience of epoch restricted to 1 and monitoring parameter equal to loss in validation set.*



```
60000/60000 [==============================] - 42s 703us/sample - loss: 0.1252 - acc: 0.9639
Epoch 14/25
60000/60000 [==============================] - 42s 705us/sample - loss: 0.1191 - acc: 0.9653
Epoch 15/25
60000/60000 [==============================] - 42s 704us/sample - loss: 0.1137 - acc: 0.9667
Epoch 16/25
60000/60000 [==============================] - 41s 689us/sample - loss: 0.1082 - acc: 0.9679
Epoch 17/25
60000/60000 [==============================] - 42s 697us/sample - loss: 0.1042 - acc: 0.9691
Epoch 18/25
60000/60000 [==============================] - 42s 693us/sample - loss: 0.0997 - acc: 0.9711
Epoch 19/25
60000/60000 [==============================] - 42s 695us/sample - loss: 0.0956 - acc: 0.9721
Epoch 20/25
60000/60000 [==============================] - 42s 696us/sample - loss: 0.0925 - acc: 0.9730
Epoch 21/25
60000/60000 [==============================] - 42s 699us/sample - loss: 0.0892 - acc: 0.9734
Epoch 22/25
60000/60000 [==============================] - 42s 704us/sample - loss: 0.0861 - acc: 0.9747
Epoch 23/25
60000/60000 [==============================] - 42s 698us/sample - loss: 0.0832 - acc: 0.9755
Epoch 24/25
60000/60000 [==============================] - 42s 702us/sample - loss: 0.0803 - acc: 0.9766
Epoch 25/25
60000/60000 [==============================] - 41s 691us/sample - loss: 0.0774 - acc: 0.9772
```

Fig17: *Output without Early Stopping*

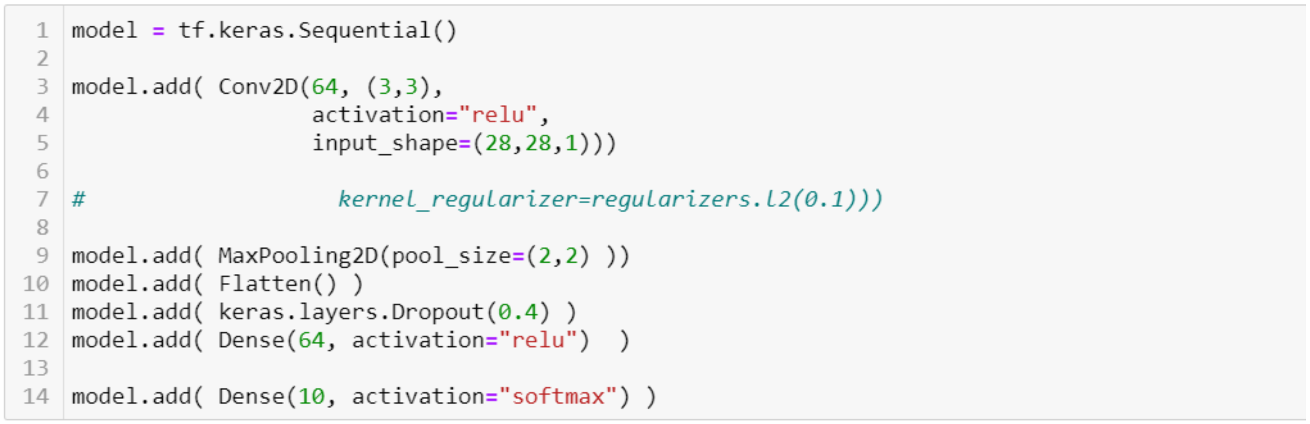


```
60000/60000 [==============================] - 46s 766us/sample - loss: 0.1914 - acc: 0.9438 - val_loss: 0.1774 - val_acc: 0.94
87
Epoch 8/25
60000/60000 [==============================] - 46s 759us/sample - loss: 0.1771 - acc: 0.9485 - val_loss: 0.1629 - val_acc: 0.95
22
Epoch 9/25
60000/60000 [==============================] - 46s 763us/sample - loss: 0.1645 - acc: 0.9517 - val_loss: 0.1532 - val_acc: 0.95
52
Epoch 10/25
60000/60000 [==============================] - 46s 766us/sample - loss: 0.1537 - acc: 0.9553 - val_loss: 0.1482 - val_acc: 0.95
67
Epoch 11/25
60000/60000 [==============================] - 46s 768us/sample - loss: 0.1445 - acc: 0.9581 - val_loss: 0.1382 - val_acc: 0.95
96
Epoch 12/25
60000/60000 [==============================] - 46s 768us/sample - loss: 0.1355 - acc: 0.9605 - val_loss: 0.1307 - val_acc: 0.96
15
Epoch 13/25
60000/60000 [==============================] - 44s 726us/sample - loss: 0.1284 - acc: 0.9623 - val_loss: 0.1244 - val_acc: 0.96
22
Epoch 14/25
60000/60000 [==============================] - 43s 724us/sample - loss: 0.1210 - acc: 0.9650 - val_loss: 0.1358 - val_acc: 0.95
84

t[419]: <tensorflow.python.keras.callbacks.History at 0x1be64bf87b8>

[420]:  test_loss, test_acc = model.evaluate(test_images, test_labels)
        print(test_acc)

10000/10000 [==============================] - 2s 183us/sample - loss: 0.1358 - acc: 0.9584
0.9584
```

Fig18: *Output with early stopping implementation*

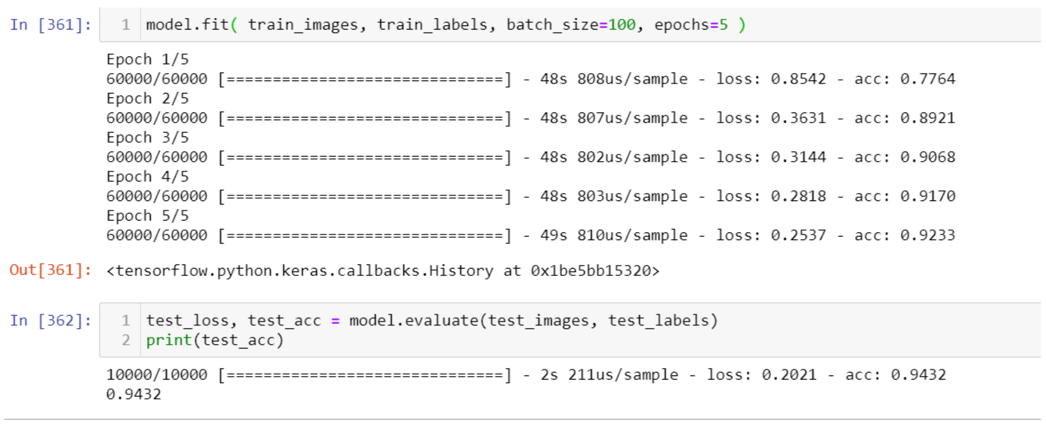
It is worth noting that the epochs are reduced to half saving a lot of computational steps for training after the early stopping regularization method is implemented.

Dropout: This technique in comparison to the previous ones show the best result. The training efficiency decreases as expected however the testing accuracy observes a slight raise that the general method without regularization. The dropout probability is kept to '40%'.The below figures show the implementation and the outputs thus observed:



```python
model = tf.keras.Sequential()

model.add( Conv2D(64, (3,3),
                  activation="relu",
                  input_shape=(28,28,1)))

#                    kernel_regularizer=regularizers.l2(0.1)))

model.add( MaxPooling2D(pool_size=(2,2) ))
model.add( Flatten() )
model.add( keras.layers.Dropout(0.4) )
model.add( Dense(64, activation="relu")  )

model.add( Dense(10, activation="softmax") )
```

Fig19: *Dropout implementation*



```
In [361]:  model.fit( train_images, train_labels, batch_size=100, epochs=5 )

           Epoch 1/5
           60000/60000 [==============================] - 48s 808us/sample - loss: 0.8542 - acc: 0.7764
           Epoch 2/5
           60000/60000 [==============================] - 48s 807us/sample - loss: 0.3631 - acc: 0.8921
           Epoch 3/5
           60000/60000 [==============================] - 48s 802us/sample - loss: 0.3144 - acc: 0.9068
           Epoch 4/5
           60000/60000 [==============================] - 48s 803us/sample - loss: 0.2818 - acc: 0.9170
           Epoch 5/5
           60000/60000 [==============================] - 49s 810us/sample - loss: 0.2537 - acc: 0.9233

Out[361]:  <tensorflow.python.keras.callbacks.History at 0x1be5bb15320>

In [362]:  test_loss, test_acc = model.evaluate(test_images, test_labels)
           print(test_acc)

           10000/10000 [==============================] - 2s 211us/sample - loss: 0.2021 - acc: 0.9432
           0.9432
```

Fig20: *Output with Dropout Regularization*

According to the above all discussed regularizations the Dropout method of regularization turns out to be the best depending on our given training model and the dataset. L1 and L2 show similar trends in training datasets however the testing efficiency is drastically compromised in case of L1. Early Stopping technique results in good results but the computational power and time consumed is way too much while dropout gives almost similar efficiency with very less computational time and power.