Artificial Neural Networks

COMP 8610 W2020

Assignment II

Group Members:

1.	Adarsh Sai Gupta:	105181433
2.	Anshul Sharma:	110016388
3.	Rishav Chatterjee:	110010348

Question 1

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

Use your favorite deep learning platform. Marvin, Caffe (http://caffe.berkeleyvision.org), TensorFlow (https://www.tensorflow.org), Theano, Torch, Lasagne. For more DL platforms, see the link: http://deeplearning.net/software_links/.

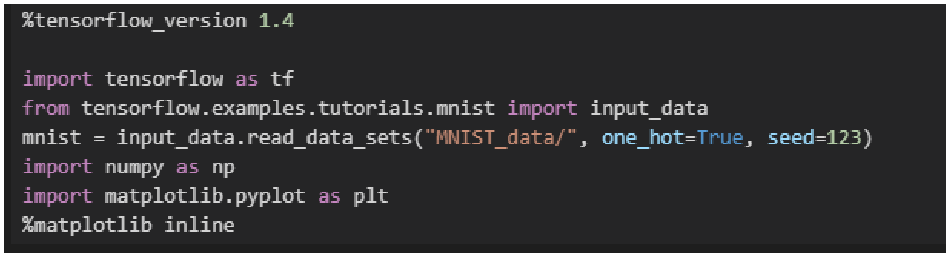
Comments: MNIST is a standard dataset for machine learning and deep learning. It's good to try it on one shallow neural network before deep neural networks. Downloading the dataset from other places in preprocessed format is allowed, but practicing how to read the dataset prepares you for other new datasets you may be interested in.

- 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. it will be very useful for you to read Chapter-11 of the textbook.

- 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. Which methods you want to try and how many you want to try and compare is up to you and up to the amount of time you have left to complete the assignment. Remember, this is a research course. You may want to read Chapter-8, which I will cover this week.

Answer:

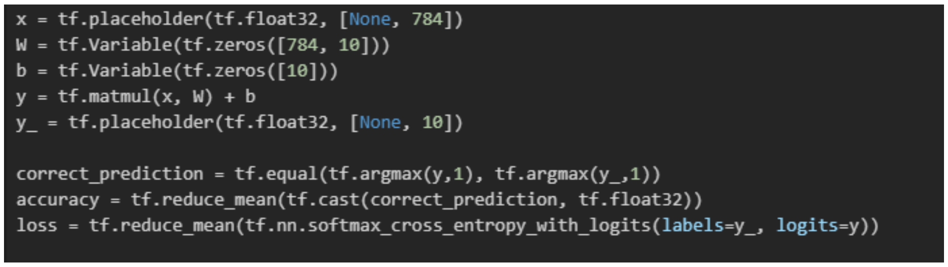
First, we import the **MNIST dataset** from Tensorflow library as follows,



```
%tensorflow_version 1.4

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True, seed=123)
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Figure 1: Importing MNIST Dataset from Tensorflow Library

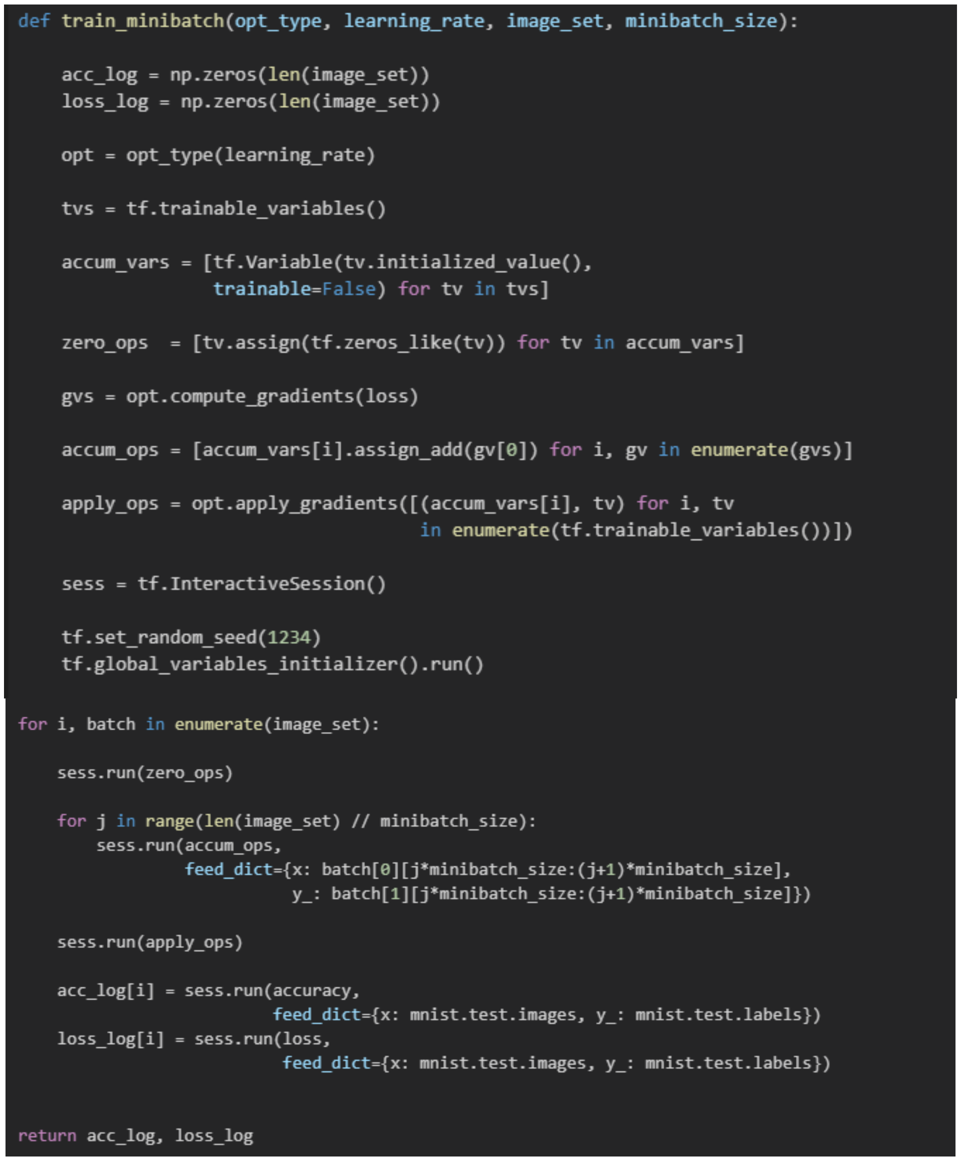
Then we create, initialize and assign the required placeholders, variables and define the *correct_prediction*, *accuracy* and *loss* variables to further train the minibatch. The dataset is split into Train and Test dataset and preprocessed for computation. **SGD** is being used for training the data. We are using **Softmax** and **Cross-entropy** on the training samples to calculate the loss. The **weight** matrix creates a tensor with all elements of shape [784, 10] set to zero. The **biases** are all set to zero.



```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
y_ = tf.placeholder(tf.float32, [None, 10])

correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
```

Figure 2: Initializing Weight, Biases and Placeholders for multiclass logistic regression

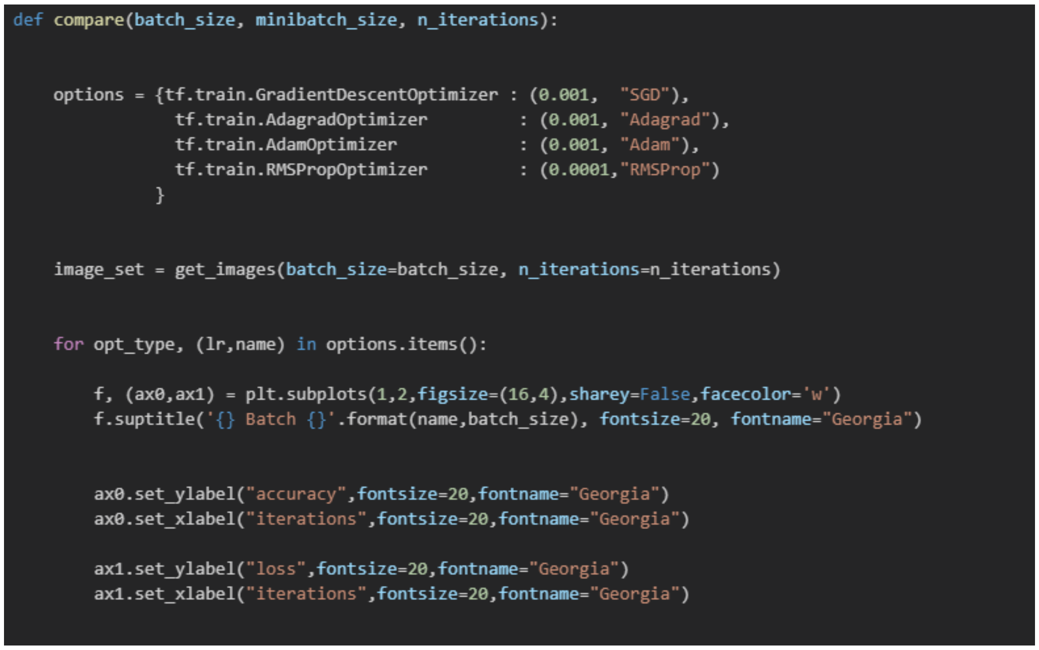
Next we train the minibatch with different hyperparameters such as **batch size** and **learning rates** etc.



```
def train_minibatch(opt_type, learning_rate, image_set, minibatch_size):

    acc_log = np.zeros(len(image_set))
    loss_log = np.zeros(len(image_set))

    opt = opt_type(learning_rate)

    tvs = tf.trainable_variables()

    accum_vars = [tf.Variable(tv.initialized_value(),
                  trainable=False) for tv in tvs]

    zero_ops  = [tv.assign(tf.zeros_like(tv)) for tv in accum_vars]

    gvs = opt.compute_gradients(loss)

    accum_ops = [accum_vars[i].assign_add(gv[0]) for i, gv in enumerate(gvs)]

    apply_ops = opt.apply_gradients([(accum_vars[i], tv) for i, tv
                                    in enumerate(tf.trainable_variables())])

    sess = tf.InteractiveSession()

    tf.set_random_seed(1234)
    tf.global_variables_initializer().run()

for i, batch in enumerate(image_set):

    sess.run(zero_ops)

    for j in range(len(image_set) // minibatch_size):
        sess.run(accum_ops,
                 feed_dict={x: batch[0][j*minibatch_size:(j+1)*minibatch_size],
                            y_: batch[1][j*minibatch_size:(j+1)*minibatch_size]})

    sess.run(apply_ops)

    acc_log[i] = sess.run(accuracy,
                          feed_dict={x: mnist.test.images, y_: mnist.test.labels})
    loss_log[i] = sess.run(loss,
                           feed_dict={x: mnist.test.images, y_: mnist.test.labels})


return acc_log, loss_log
```

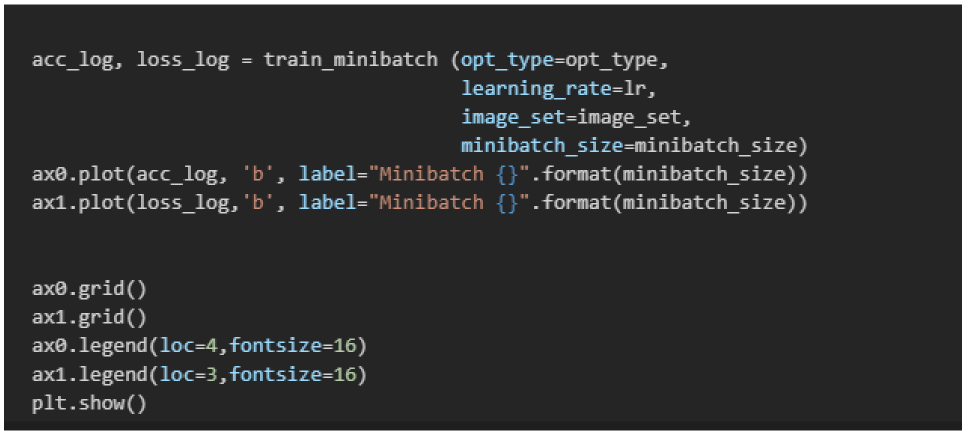
Figure 3: The method definition for training the minibatch

We also use **Gradient Decent Optimizer** to reduce the loss in the loss in training samples as seen in the above and the following figure.



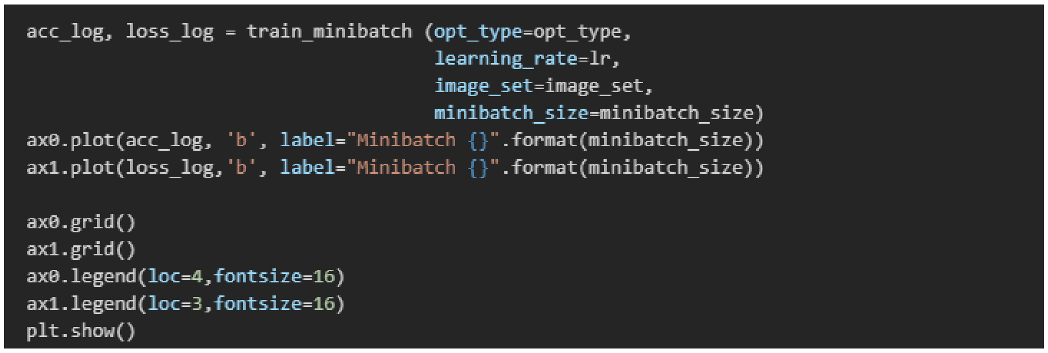
```python
def compare(batch_size, minibatch_size, n_iterations):


    options = {tf.train.GradientDescentOptimizer : (0.001,  "SGD"),
               tf.train.AdagradOptimizer         : (0.001, "Adagrad"),
               tf.train.AdamOptimizer            : (0.001, "Adam"),
               tf.train.RMSPropOptimizer         : (0.0001,"RMSProp")
              }


    image_set = get_images(batch_size=batch_size, n_iterations=n_iterations)


    for opt_type, (lr,name) in options.items():

        f, (ax0,ax1) = plt.subplots(1,2,figsize=(16,4),sharey=False,facecolor='w')
        f.suptitle('{} Batch {}'.format(name,batch_size), fontsize=20, fontname="Georgia")


        ax0.set_ylabel("accuracy",fontsize=20,fontname="Georgia")
        ax0.set_xlabel("iterations",fontsize=20,fontname="Georgia")

        ax1.set_ylabel("loss",fontsize=20,fontname="Georgia")
        ax1.set_xlabel("iterations",fontsize=20,fontname="Georgia")
```

Figure 4: Defining the Compare method to show the comparison between SGD, Adam, Adagrad & RMSProp



```python
acc_log, loss_log = train_minibatch (opt_type=opt_type,
                                     learning_rate=lr,
                                     image_set=image_set,
                                     minibatch_size=minibatch_size)
ax0.plot(acc_log, 'b', label="Minibatch {}".format(minibatch_size))
ax1.plot(loss_log,'b', label="Minibatch {}".format(minibatch_size))


ax0.grid()
ax1.grid()
ax0.legend(loc=4,fontsize=16)
ax1.legend(loc=3,fontsize=16)
plt.show()
```

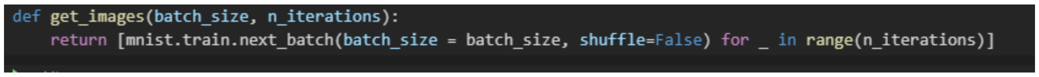
Figure 5: Executing the multiclass logistic regression



```python
acc_log, loss_log = train_minibatch (opt_type=opt_type,
                                     learning_rate=lr,
                                     image_set=image_set,
                                     minibatch_size=minibatch_size)
ax0.plot(acc_log, 'b', label="Minibatch {}".format(minibatch_size))
ax1.plot(loss_log,'b', label="Minibatch {}".format(minibatch_size))

ax0.grid()
ax1.grid()
ax0.legend(loc=4,fontsize=16)
ax1.legend(loc=3,fontsize=16)
plt.show()
```

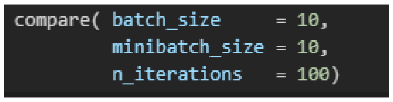
Figure 6: Plotting the results

The function for loading a fixed set of images to run the training on.



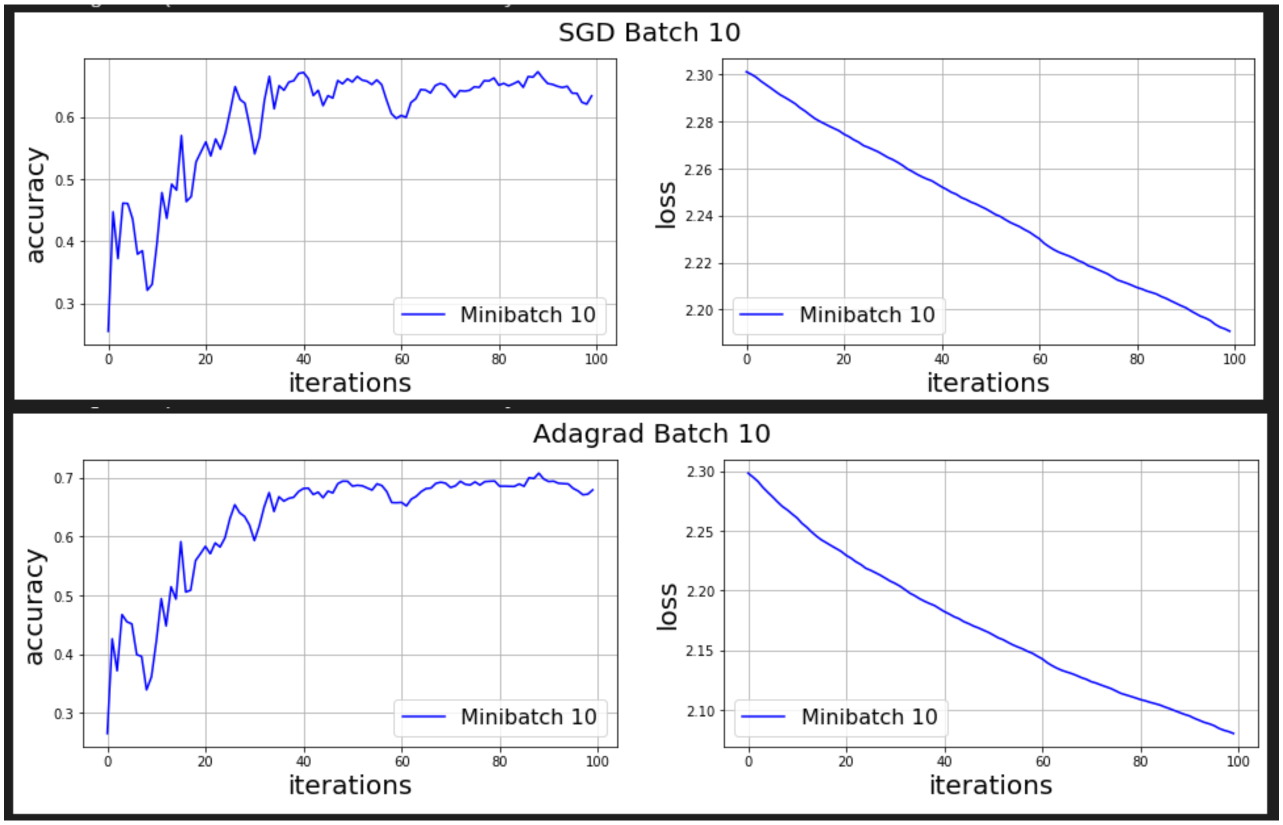
```
def get_images(batch_size, n_iterations):
    return [mnist.train.next_batch(batch_size = batch_size, shuffle=False) for _ in range(n_iterations)]
```

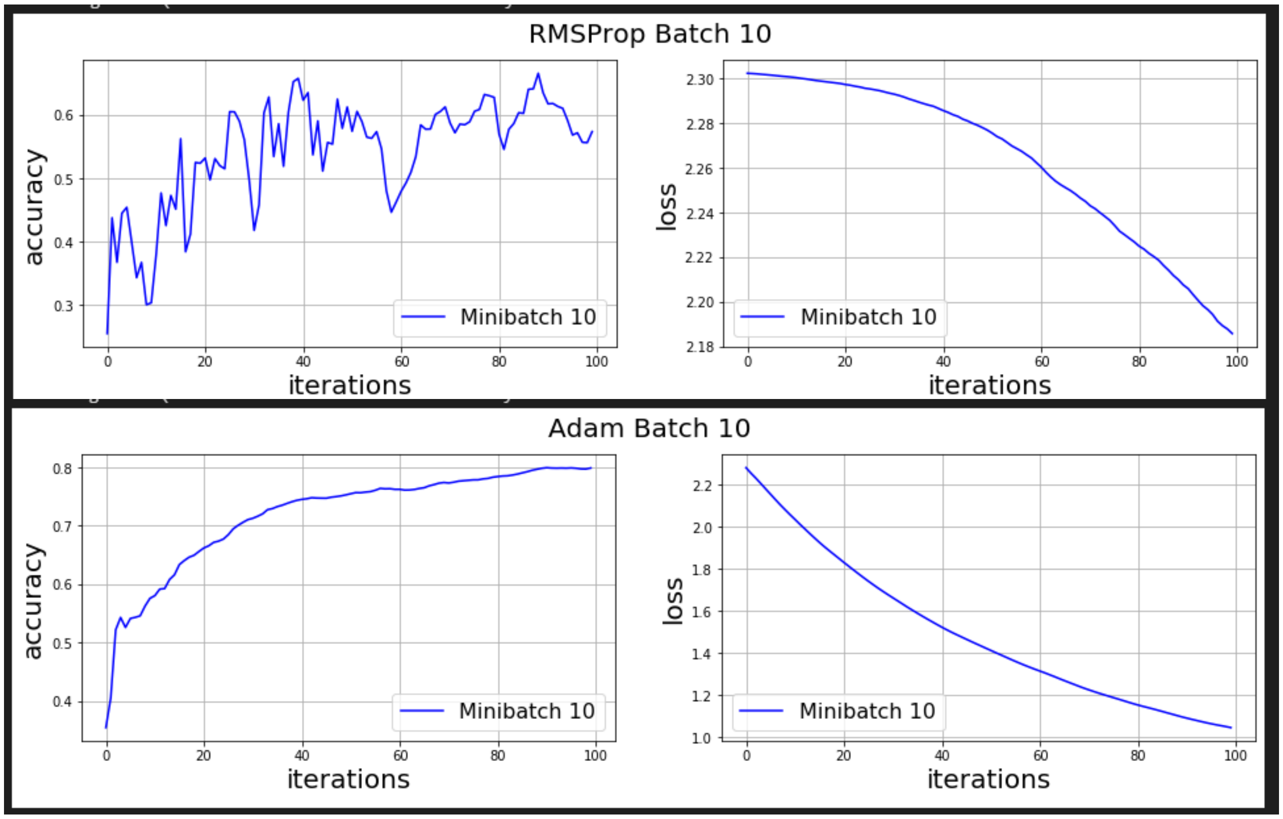
Finally, we call the compare method with different values of **batch size, minibatch size and iterations** to see the results.



```
compare( batch_size     = 10,
         minibatch_size = 10,
         n_iterations   = 100)
```

The following is a representation of the performance comparison of **minibatch SGD, Adagrad, Adam and RMSProp batch** with different hyperparameters such as batch size, learning rate and iterations.





Summary: As we can see from the above results, using **multiclass regression** and with a **batch size of 10**, **minibatch size of 10** and **100 iterations**, we get a result which illustrates that the accuracy keeps on increasing with the number of iterations. The **accuracy** for SGD seems to be around 70%, for Adam it is about 80%, for RMSProp, 70% and lastly, for Adagrad, it is about 80%. Thus, we may say that accuracy of the algorithm is directly proportional to the number of iterations. Similarly, the **loss** seems to drop with increasing number of iterations for each case. In some cases, like for SGD the drop-in loss seems to be more uniform than the one in RMSProp.

Question 2

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

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

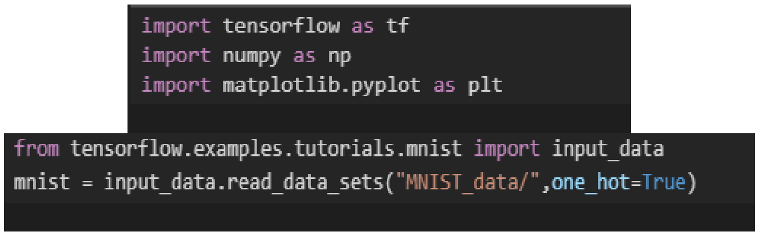
where {wi, bi} are all the parameters in the logistic regression, and λ ∈ R is the regularization hyperparameter. Typically, λ is about C/n where n is the number of data points and C is some constant in [0.01,100] (need to tune C). Run the regularized multiclass logistic regression on MNIST, using the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

Answer:

For Question 1, we implemented multi-class logistic regression on MNIST dataset with minibatch SGD. We also made comparisons with other optimization methods.

For this question we will follow a similar procedure but with **L2 regularization**.

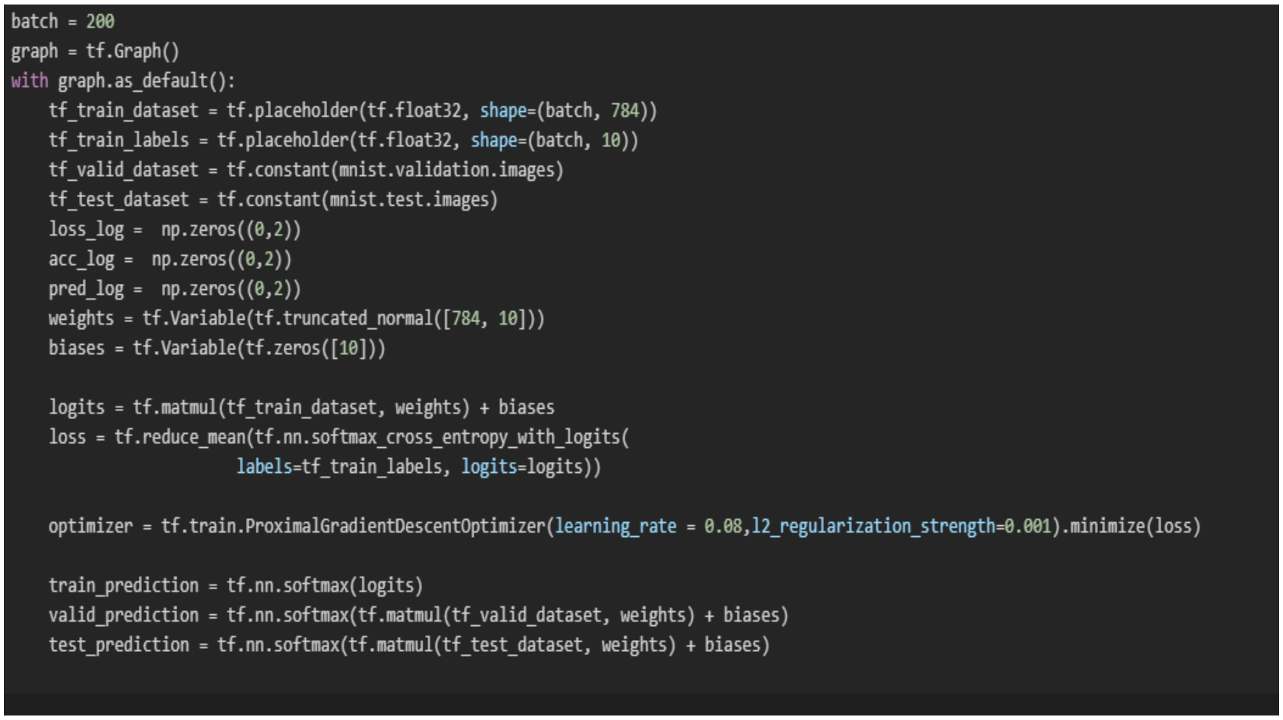
First, we implement all the dependencies. We import the dataset from Tensorflow library.



```
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)
```

Figure 7: Importing MNIST data from tensorflow library

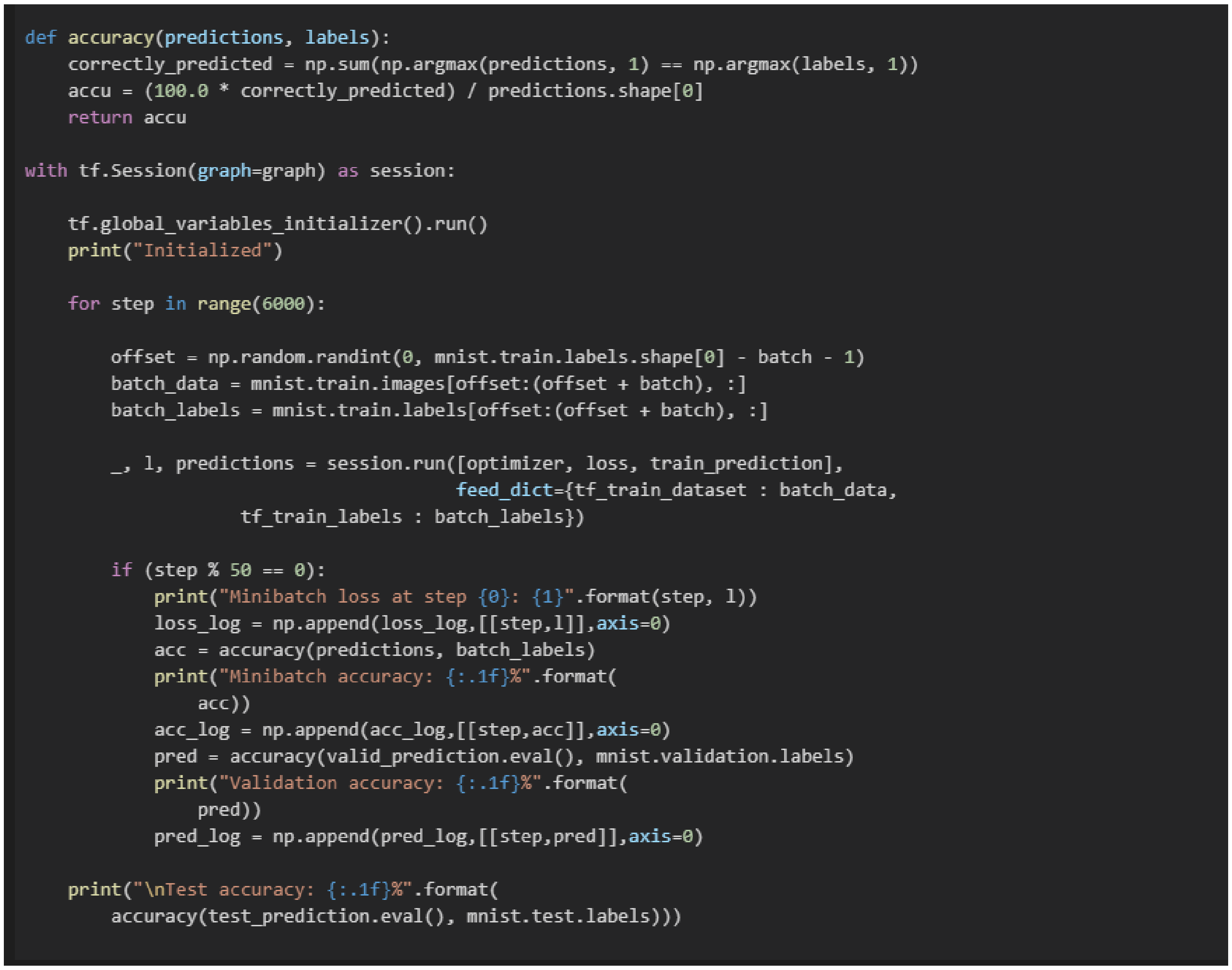
Next, prepare the data for L2 regularization and define the prediction variables **for training set, validation set and test set.**



```
batch = 200
graph = tf.Graph()
with graph.as_default():
    tf_train_dataset = tf.placeholder(tf.float32, shape=(batch, 784))
    tf_train_labels = tf.placeholder(tf.float32, shape=(batch, 10))
    tf_valid_dataset = tf.constant(mnist.validation.images)
    tf_test_dataset = tf.constant(mnist.test.images)
    loss_log =  np.zeros((0,2))
    acc_log =  np.zeros((0,2))
    pred_log =  np.zeros((0,2))
    weights = tf.Variable(tf.truncated_normal([784, 10]))
    biases = tf.Variable(tf.zeros([10]))

    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 = tf.train.ProximalGradientDescentOptimizer(learning_rate = 0.08,l2_regularization_strength=0.001).minimize(loss)

    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)
```

Figure 8: Initializing a tensorflow graph and preparing the data for L2 regularization

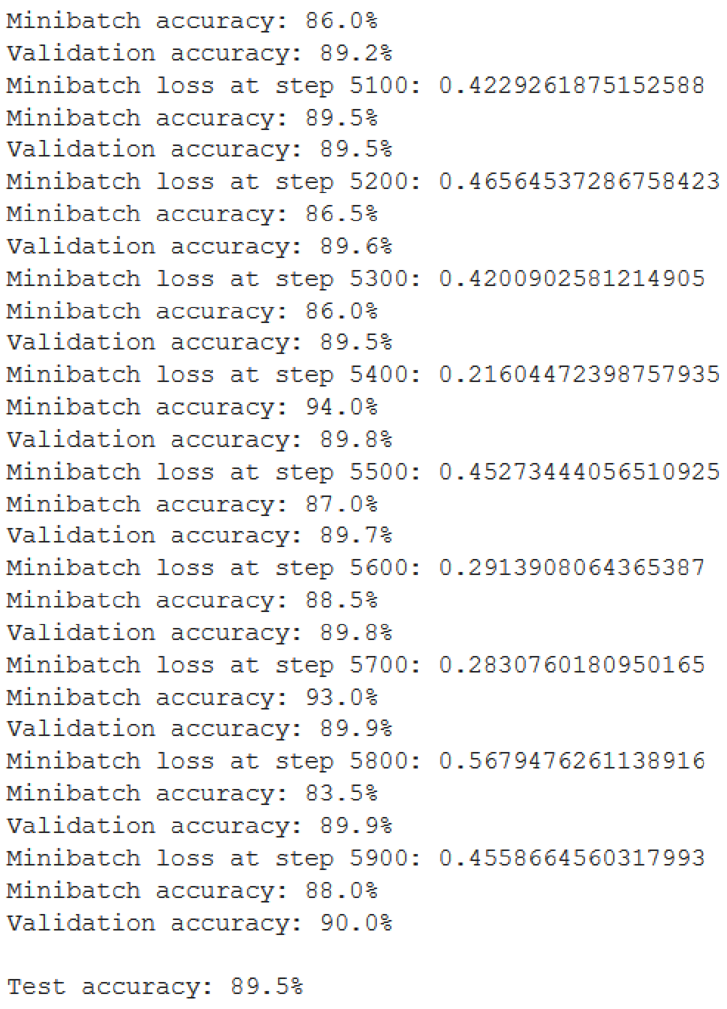
Then, we initialize the **weights and biases** and pick a **randomized offset** to generate a minibatch.



```
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.Session(graph=graph) as session:

    tf.global_variables_initializer().run()
    print("Initialized")

    for step in range(6000):

        offset = np.random.randint(0, mnist.train.labels.shape[0] - batch - 1)
        batch_data = mnist.train.images[offset:(offset + batch), :]
        batch_labels = mnist.train.labels[offset:(offset + batch), :]

        _, l, predictions = session.run([optimizer, loss, train_prediction],
                                        feed_dict={tf_train_dataset : batch_data,
                    tf_train_labels : batch_labels})

        if (step % 50 == 0):
            print("Minibatch loss at step {0}: {1}".format(step, l))
            loss_log = np.append(loss_log,[[step,l]],axis=0)
            acc = accuracy(predictions, batch_labels)
            print("Minibatch accuracy: {:.1f}%".format(
                acc))
            acc_log = np.append(acc_log,[[step,acc]],axis=0)
            pred = accuracy(valid_prediction.eval(), mnist.validation.labels)
            print("Validation accuracy: {:.1f}%".format(
                pred))
            pred_log = np.append(pred_log,[[step,pred]],axis=0)

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

Figure 9: Executing the L2 regularization

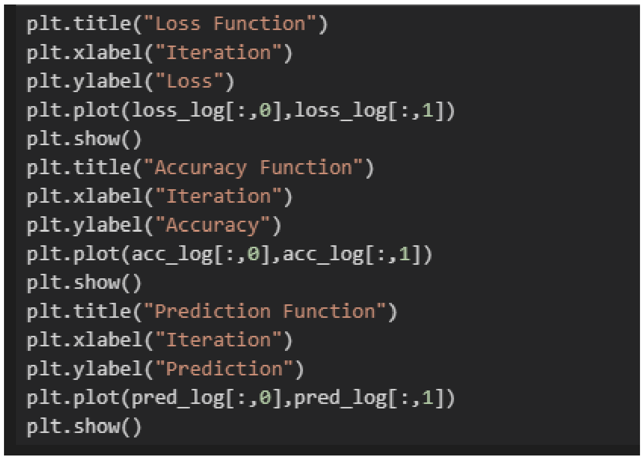
The following is the output of the L2 regularization on minibatch SGD. We can see from the results that at a step size of around 5500, the **test accuracy** is **89.5%, validation accuracy** is about **89%**, the **minibatch accuracy** is about **87%** and the **loss** is about **28%.**



```
Minibatch accuracy: 86.0%
Validation accuracy: 89.2%
Minibatch loss at step 5100: 0.4229261875152588
Minibatch accuracy: 89.5%
Validation accuracy: 89.5%
Minibatch loss at step 5200: 0.46564537286758423
Minibatch accuracy: 86.5%
Validation accuracy: 89.6%
Minibatch loss at step 5300: 0.4200902581214905
Minibatch accuracy: 86.0%
Validation accuracy: 89.5%
Minibatch loss at step 5400: 0.21604472398757935
Minibatch accuracy: 94.0%
Validation accuracy: 89.8%
Minibatch loss at step 5500: 0.45273444056510925
Minibatch accuracy: 87.0%
Validation accuracy: 89.7%
Minibatch loss at step 5600: 0.2913908064365387
Minibatch accuracy: 88.5%
Validation accuracy: 89.8%
Minibatch loss at step 5700: 0.2830760180950165
Minibatch accuracy: 93.0%
Validation accuracy: 89.9%
Minibatch loss at step 5800: 0.5679476261138916
Minibatch accuracy: 83.5%
Validation accuracy: 89.9%
Minibatch loss at step 5900: 0.4558664560317993
Minibatch accuracy: 88.0%
Validation accuracy: 90.0%

Test accuracy: 89.5%
```

Figure 10: Output of L2 Regularization

Lastly, for further clarification, we generate the graphs for L2 Regularization on minibatch SGD.



```
plt.title("Loss Function")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.plot(loss_log[:,0],loss_log[:,1])
plt.show()
plt.title("Accuracy Function")
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.plot(acc_log[:,0],acc_log[:,1])
plt.show()
plt.title("Prediction Function")
plt.xlabel("Iteration")
plt.ylabel("Prediction")
plt.plot(pred_log[:,0],pred_log[:,1])
plt.show()
```

Figure 11: Plotting the graphs

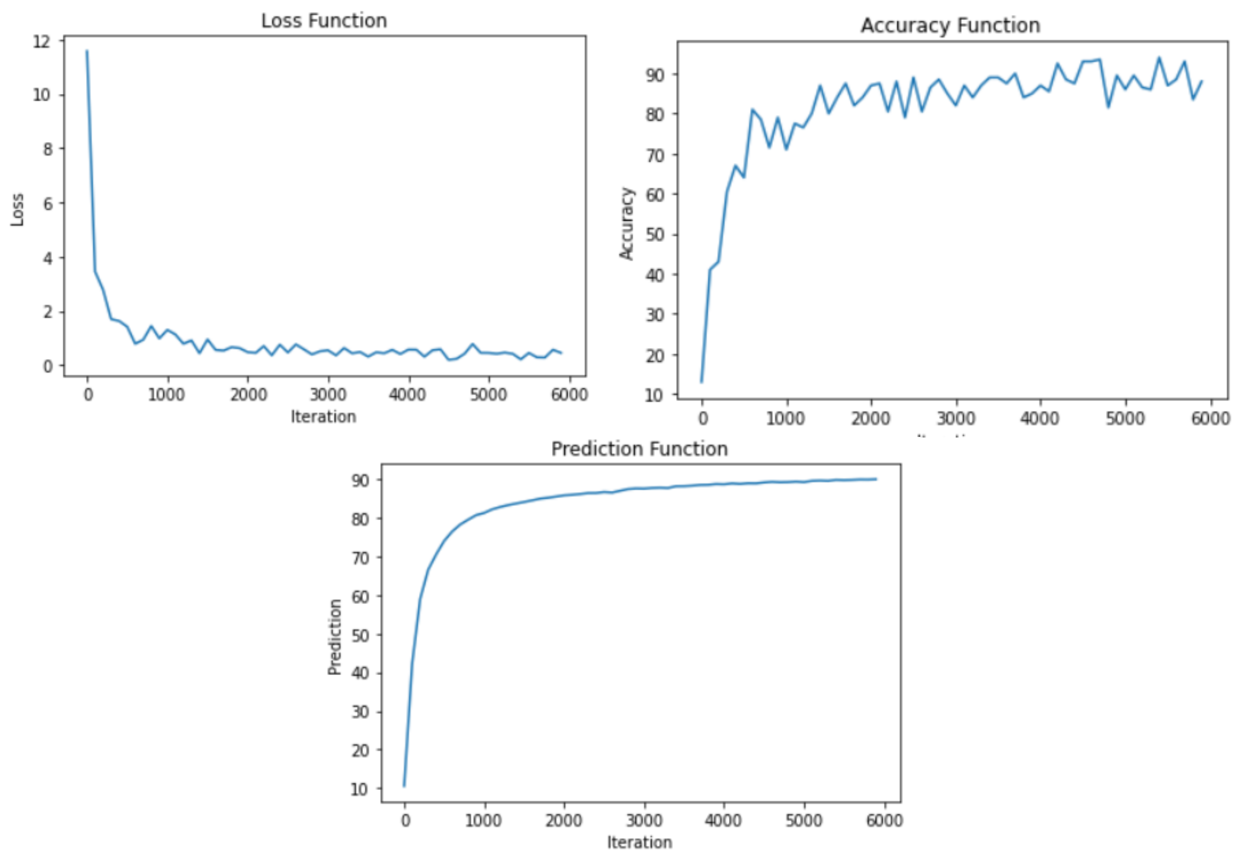


Figure 12: Loss, Accuracy & Prediction against Iterations for L2

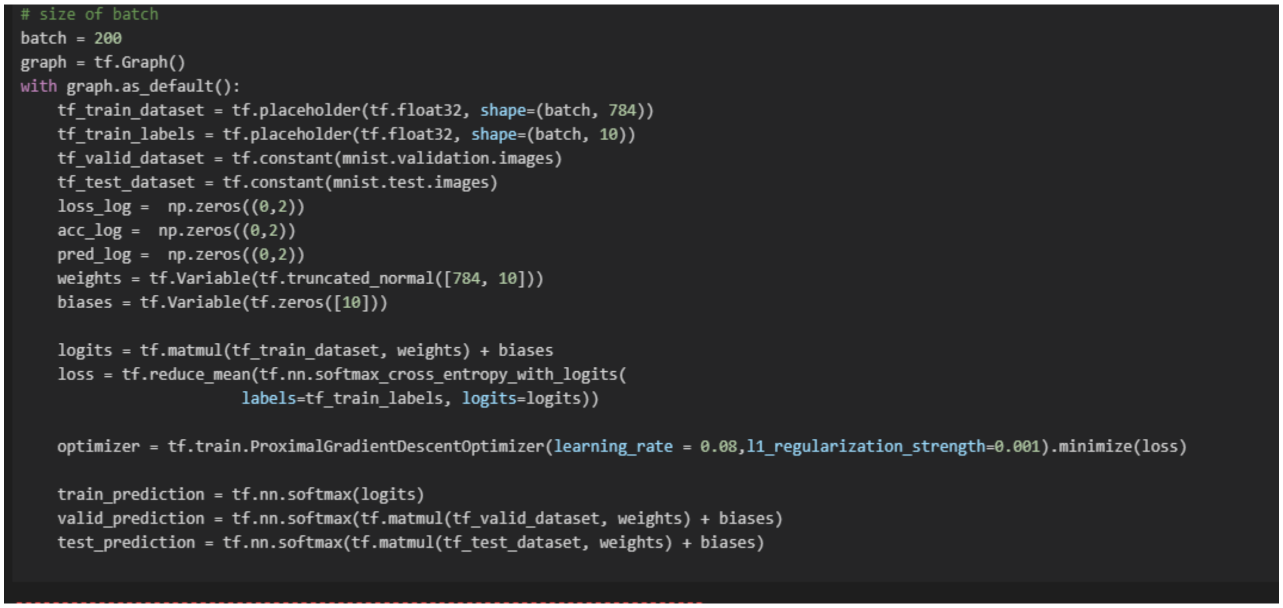
Question 3

Going above and beyond Question-1 and Question-2, investigate the basic minibatch SGD with, at least, another regularization method discussed in class (L1, data augmentation, noise robustness, early stopping, sparse representation, bagging, or dropout). Currently, L2 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.

Answer:

Question 3 is simply an extension of Question 1 & 2, where we try to implement different regularization techniques to compare their respective outputs. Hence, we implement **L1 regularization** method, a combination of **L1&L2 regularization** and **CNN** on the same dataset with minibatch SGD.

- The procedure we use to implement L1 is same as the one for L2. Implementation cell for **L1 regularization** is as follows:



```
# size of batch
batch = 200
graph = tf.Graph()
with graph.as_default():
    tf_train_dataset = tf.placeholder(tf.float32, shape=(batch, 784))
    tf_train_labels = tf.placeholder(tf.float32, shape=(batch, 10))
    tf_valid_dataset = tf.constant(mnist.validation.images)
    tf_test_dataset = tf.constant(mnist.test.images)
    loss_log =  np.zeros((0,2))
    acc_log =  np.zeros((0,2))
    pred_log =  np.zeros((0,2))
    weights = tf.Variable(tf.truncated_normal([784, 10]))
    biases = tf.Variable(tf.zeros([10]))

    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 = tf.train.ProximalGradientDescentOptimizer(learning_rate = 0.08,l1_regularization_strength=0.001).minimize(loss)

    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)
```

Figure 13: Initializing a tensorflow graph and preparing the data for L1 regularization

```
Validation accuracy: 88.6%
Minibatch loss at step 5000: 0.2555373013019562
Minibatch accuracy: 91.5%
Validation accuracy: 88.8%
Minibatch loss at step 5100: 0.4105415642261505
Minibatch accuracy: 87.5%
Validation accuracy: 89.1%
Minibatch loss at step 5200: 0.3811723291873932
Minibatch accuracy: 90.0%
Validation accuracy: 89.1%
Minibatch loss at step 5300: 0.5998455882072449
Minibatch accuracy: 84.0%
Validation accuracy: 89.0%
Minibatch loss at step 5400: 0.3656597137451172
Minibatch accuracy: 89.0%
Validation accuracy: 89.1%
Minibatch loss at step 5500: 0.42139723896980286
Minibatch accuracy: 87.5%
Validation accuracy: 89.2%
Minibatch loss at step 5600: 0.28818708658218384
Minibatch accuracy: 91.0%
Validation accuracy: 89.2%
Minibatch loss at step 5700: 0.3857038915157318
Minibatch accuracy: 87.0%
Validation accuracy: 89.3%
Minibatch loss at step 5800: 0.433034211397171
Minibatch accuracy: 88.0%
Validation accuracy: 89.4%
Minibatch loss at step 5900: 0.24028171598911285
Minibatch accuracy: 93.0%
Validation accuracy: 89.6%

Test accuracy: 88.9%
```

Figure 14: Output for L1 regularization

As we can see from the image above, at a step size of around 5500, the **test accuracy** is **88.9%, validation**

accuracy is about **89.2%**, the **minibatch accuracy** is about **87.5%** and the **loss** is on average **30%.**
We further plot the graph for further clarification using the same procedure as for L2.

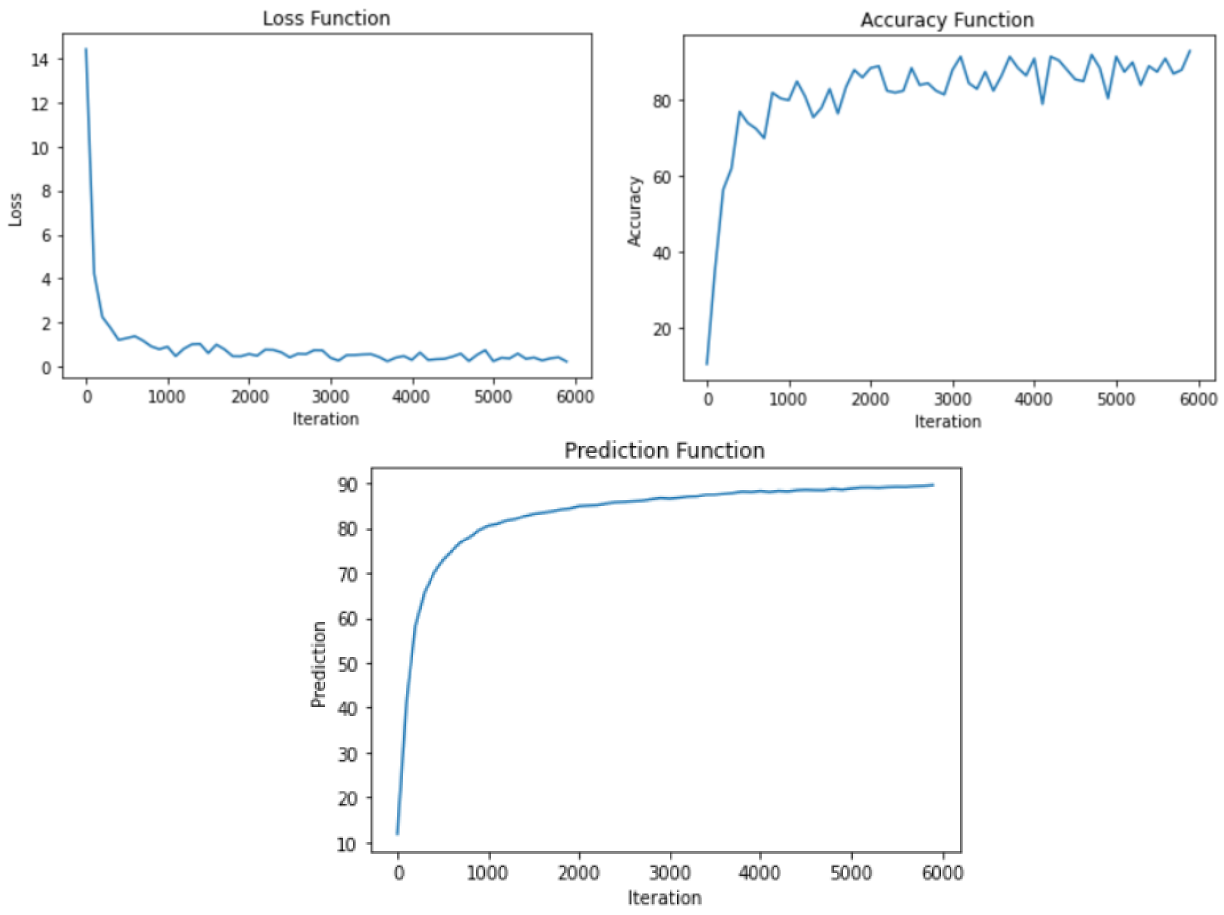
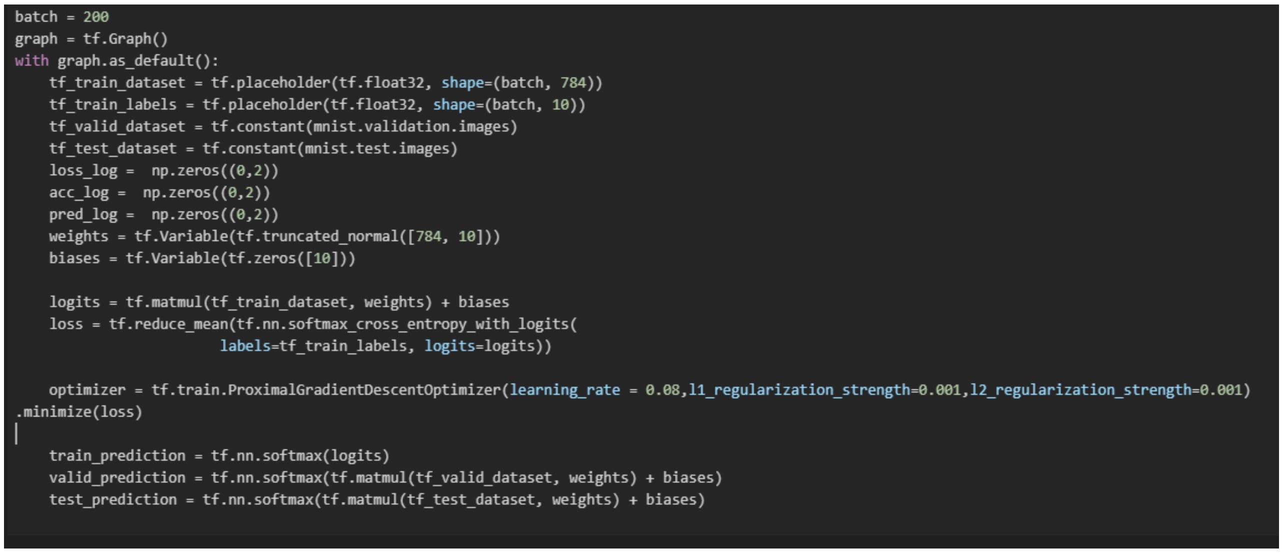


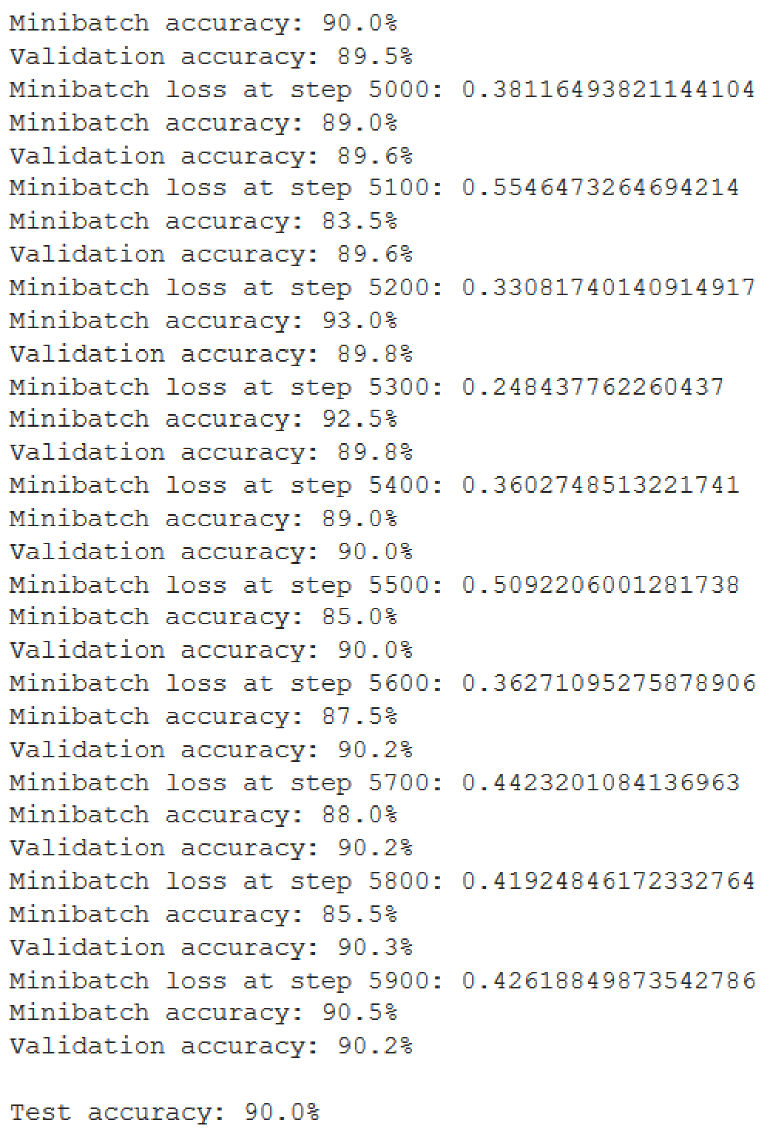
Figure 15: Loss, Accuracy & Prediction against Iterations L1 regularization

- Implementation for combined **L1&L2** regularization:



```
batch = 200
graph = tf.Graph()
with graph.as_default():
    tf_train_dataset = tf.placeholder(tf.float32, shape=(batch, 784))
    tf_train_labels = tf.placeholder(tf.float32, shape=(batch, 10))
    tf_valid_dataset = tf.constant(mnist.validation.images)
    tf_test_dataset = tf.constant(mnist.test.images)
    loss_log =  np.zeros((0,2))
    acc_log =  np.zeros((0,2))
    pred_log =  np.zeros((0,2))
    weights = tf.Variable(tf.truncated_normal([784, 10]))
    biases = tf.Variable(tf.zeros([10]))

    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 = tf.train.ProximalGradientDescentOptimizer(learning_rate = 0.08,l1_regularization_strength=0.001,l2_regularization_strength=0.001)
.minimize(loss)
|
    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)
```

Figure 16: Initializing a tensorflow graph and preparing the data for L1&L2 regulariztion



```
Minibatch accuracy: 90.0%
Validation accuracy: 89.5%
Minibatch loss at step 5000: 0.38116493821144104
Minibatch accuracy: 89.0%
Validation accuracy: 89.6%
Minibatch loss at step 5100: 0.5546473264694214
Minibatch accuracy: 83.5%
Validation accuracy: 89.6%
Minibatch loss at step 5200: 0.33081740140914917
Minibatch accuracy: 93.0%
Validation accuracy: 89.8%
Minibatch loss at step 5300: 0.248437762260437
Minibatch accuracy: 92.5%
Validation accuracy: 89.8%
Minibatch loss at step 5400: 0.3602748513221741
Minibatch accuracy: 89.0%
Validation accuracy: 90.0%
Minibatch loss at step 5500: 0.5092206001281738
Minibatch accuracy: 85.0%
Validation accuracy: 90.0%
Minibatch loss at step 5600: 0.36271095275878906
Minibatch accuracy: 87.5%
Validation accuracy: 90.2%
Minibatch loss at step 5700: 0.4423201084136963
Minibatch accuracy: 88.0%
Validation accuracy: 90.2%
Minibatch loss at step 5800: 0.41924846172332764
Minibatch accuracy: 85.5%
Validation accuracy: 90.3%
Minibatch loss at step 5900: 0.42618849873542786
Minibatch accuracy: 90.5%
Validation accuracy: 90.2%

Test accuracy: 90.0%
```

Figure 17: Output of L1&L2

From the output above, at a step size of around 5500, the **test accuracy** is **90.0%, validation accuracy** is about **900%**, the **minibatch accuracy** is about **85.0%** and the **loss** is on average **34%.**
We further plot the graph for further clarification using the same procedure as for L1&L2.

The also plot the graph to see the slope of loss, accuracy and predication.

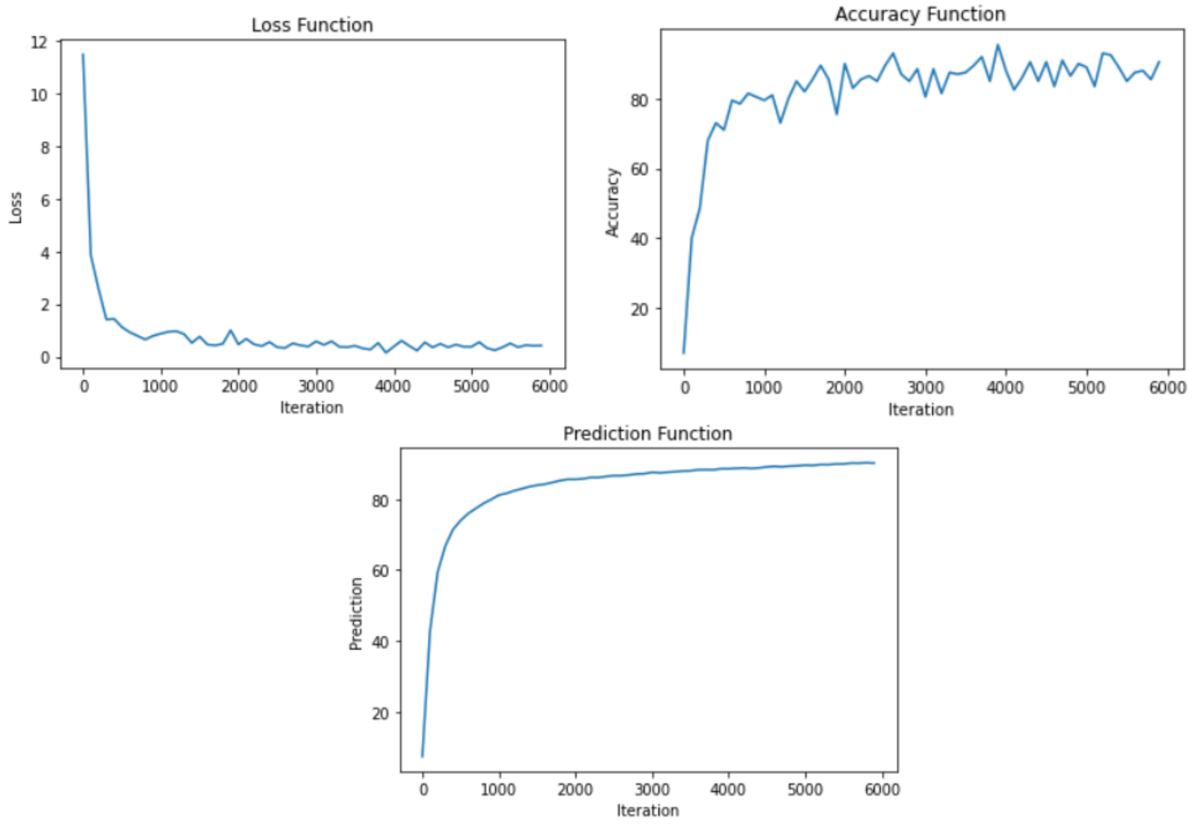
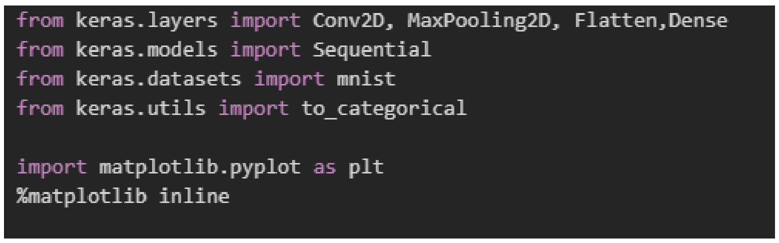


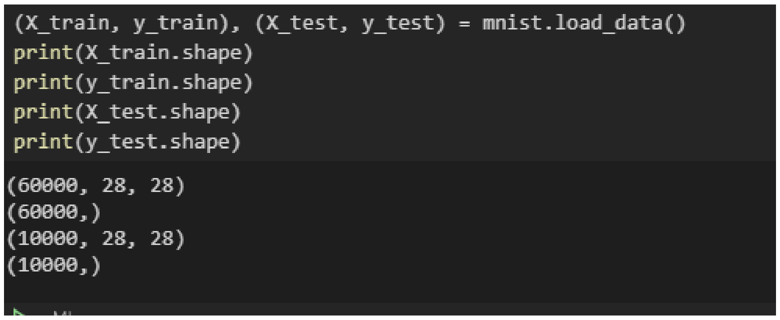
Figure 18: Loss, Accuracy & Prediction against Iterations L1&L2 regularization

- Implementation for **CNN** is as follows,



```
from keras.layers import Conv2D, MaxPooling2D, Flatten,Dense
from keras.models import Sequential
from keras.datasets import mnist
from keras.utils import to_categorical

import matplotlib.pyplot as plt
%matplotlib inline
```

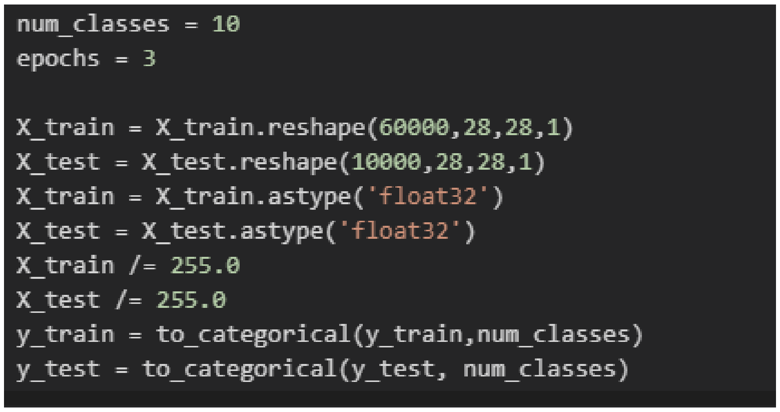
First, we load the train and test data from MNIST and verify the shape of the feature set.



```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

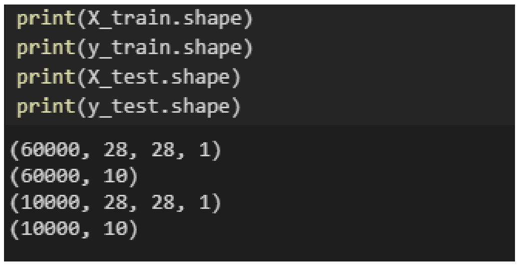
Then, we define the epochs and do the data preprocessing.



```
num_classes = 10
epochs = 3

X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255.0
X_test /= 255.0
y_train = to_categorical(y_train,num_classes)
y_test = to_categorical(y_test, num_classes)
```

Figure 19: Splitting the dataset into test and train set

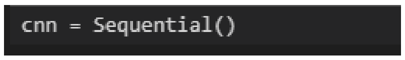
After preprocessing, we are verifying the shape of the feature set by printing the results.



```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

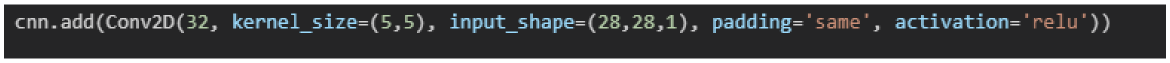
```
(60000, 28, 28, 1)
(60000, 10)
(10000, 28, 28, 1)
(10000, 10)
```

Next we are initializing the CNN sequential model



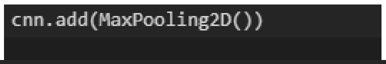
```
cnn = Sequential()
```

We are using Relu as activation function and inputting necessary variables and adding layers to the CNN model.

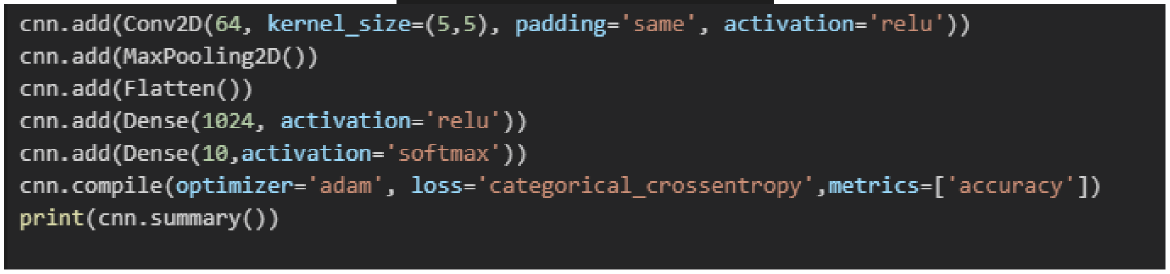


```
cnn.add(Conv2D(32, kernel_size=(5,5), input_shape=(28,28,1), padding='same', activation='relu'))
```

As we can see, we are using **Relu activation function** for the 1024 layers and then using **softmax activation** function on 10 layers. The **optimizer** we are using is **Adam** and the **loss function** is **cross-entropy.**

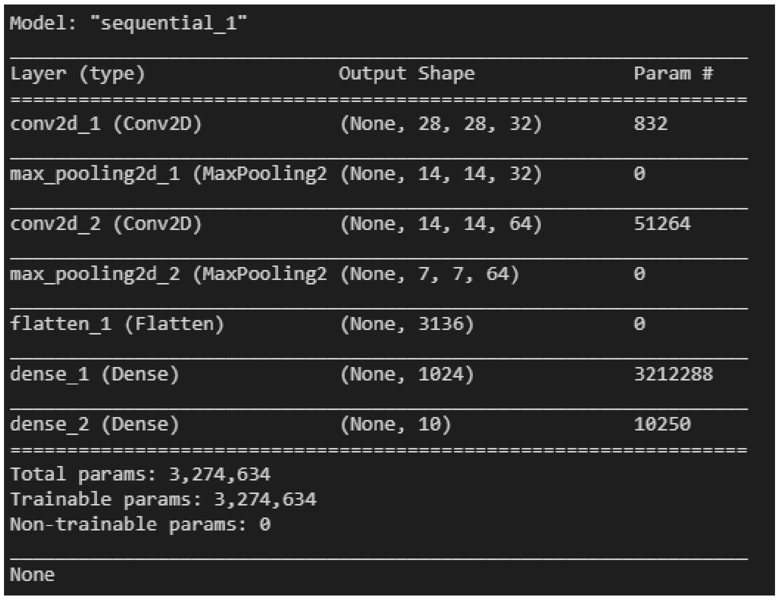


```
cnn.add(MaxPooling2D())
```



```
cnn.add(Conv2D(64, kernel_size=(5,5), padding='same', activation='relu'))
cnn.add(MaxPooling2D())
cnn.add(Flatten())
cnn.add(Dense(1024, activation='relu'))
cnn.add(Dense(10,activation='softmax'))
cnn.compile(optimizer='adam', loss='categorical_crossentropy',metrics=['accuracy'])
print(cnn.summary())
```

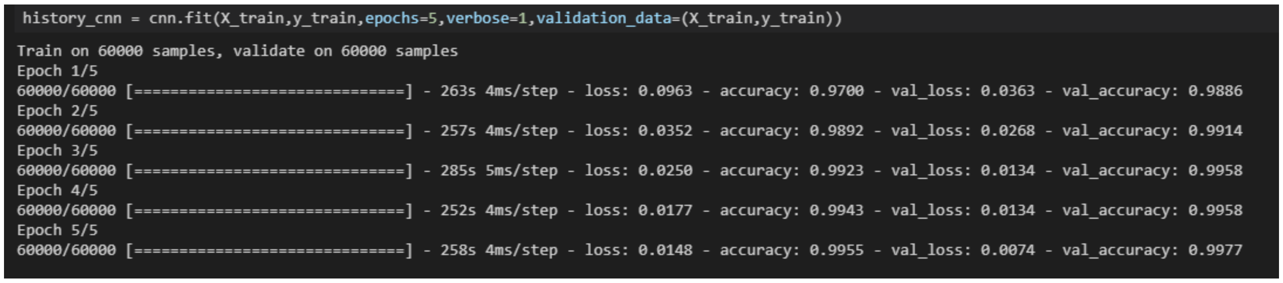
The following image illustrates the summary of the CNN model.



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              3212288
_________________________________________________________________
dense_2 (Dense)              (None, 10)                10250
=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
_________________________________________________________________
None
```

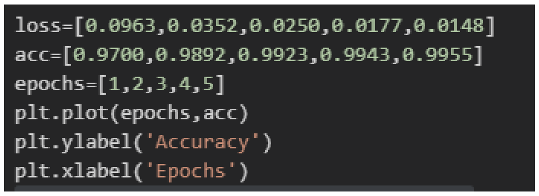
Figure 20: Summary of CNN Model

We are printing out the results for 60000 data samples and using 5 epochs. The **average accuracy** of the model seems to be **98.4%** and the **validation accuracy average** seems to be **99.2%.**



```
history_cnn = cnn.fit(X_train,y_train,epochs=5,verbose=1,validation_data=(X_train,y_train))

Train on 60000 samples, validate on 60000 samples
Epoch 1/5
60000/60000 [==============================] - 263s 4ms/step - loss: 0.0963 - accuracy: 0.9700 - val_loss: 0.0363 - val_accuracy: 0.9886
Epoch 2/5
60000/60000 [==============================] - 257s 4ms/step - loss: 0.0352 - accuracy: 0.9892 - val_loss: 0.0268 - val_accuracy: 0.9914
Epoch 3/5
60000/60000 [==============================] - 285s 5ms/step - loss: 0.0250 - accuracy: 0.9923 - val_loss: 0.0134 - val_accuracy: 0.9958
Epoch 4/5
60000/60000 [==============================] - 252s 4ms/step - loss: 0.0177 - accuracy: 0.9943 - val_loss: 0.0134 - val_accuracy: 0.9958
Epoch 5/5
60000/60000 [==============================] - 258s 4ms/step - loss: 0.0148 - accuracy: 0.9955 - val_loss: 0.0074 - val_accuracy: 0.9977
```

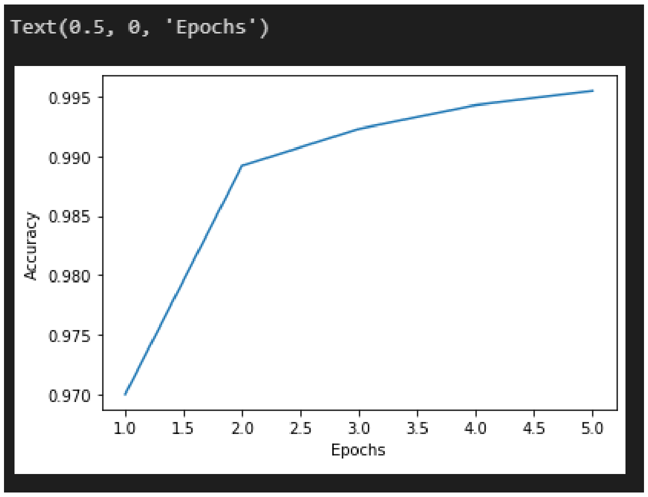
Figure 21: Output of the CNN model



```
loss=[0.0963,0.0352,0.0250,0.0177,0.0148]
acc=[0.9700,0.9892,0.9923,0.9943,0.9955]
epochs=[1,2,3,4,5]
plt.plot(epochs,acc)
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
```

Figure 22: Plotting the graph

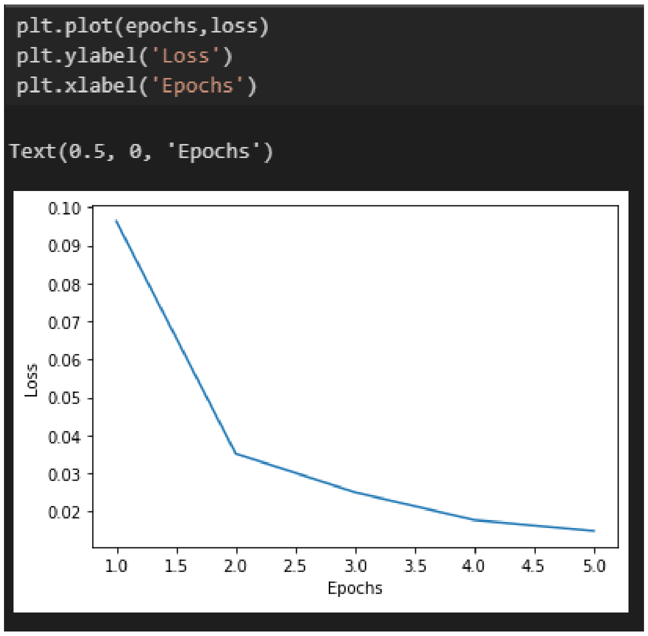
The following graph illustrates the accuracy of the model with increasing number of epochs.



```
Text(0.5, 0, 'Epochs')
```

Figure 23: Output for Accuracy vs Epochs

The following graph illustrates the loss of the model with increasing number of epochs. As per our experiment, after 4 epochs, the slope of loss gets reduced and becomes stagnant.



```
plt.plot(epochs,loss)
plt.ylabel('Loss')
plt.xlabel('Epochs')
```

```
Text(0.5, 0, 'Epochs')
```

Figure 24: Output for Loss vs Epochs

Conclusion: According to all the comparisons discussed above, CNN model seems to produce the best results with 98.4% average accuracy and 99.2% validation accuracy, which is significantly higher than L1 and combined L1&L2. The loss function for CNN also seems to produce far lower values compared to the other models. However, all the models implemented in the above comparison, yields accuracy of around or greater than 90%.

References:

[1] https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/l1
[2] https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/l2
[3] https://iq.opengenus.org/logistic-regression-tensorflow-python/
[4] https://stackoverflow.com/questions/56907971/logistic-regression-using-tensorflow-2-0
[5] https://medium.com/coinmonks/stochastic-vs-mini-batch-training-in-machine-learning-using-tensorflow-and-python-7f9709143ee2
[6] https://mmuratarat.github.io/2018-12-21/cross-entropy
