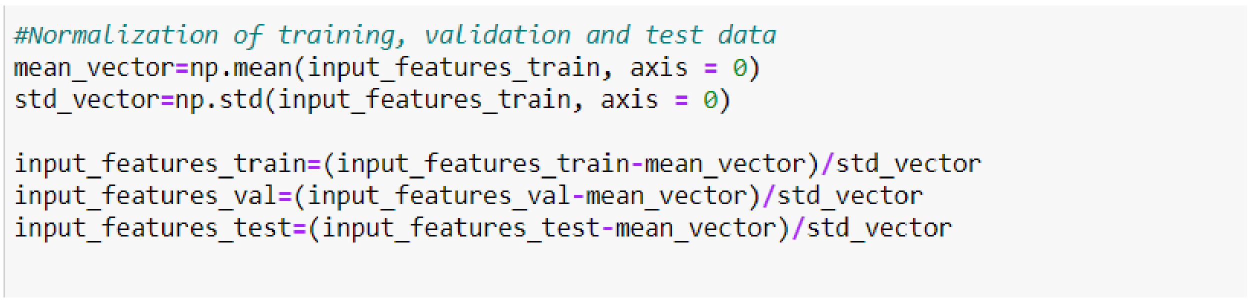
PROJECT PART 2 REPORT

Task 1 results

- **Properties of the implemented MLP (2-nH-1 MLP):**

  Activation function: Sigmoid
  Loss Used : MSE
  Learning Method : Batch
  Learning Rate : 0.0001

- **Feature Normalization** is performed on training, validation and test data



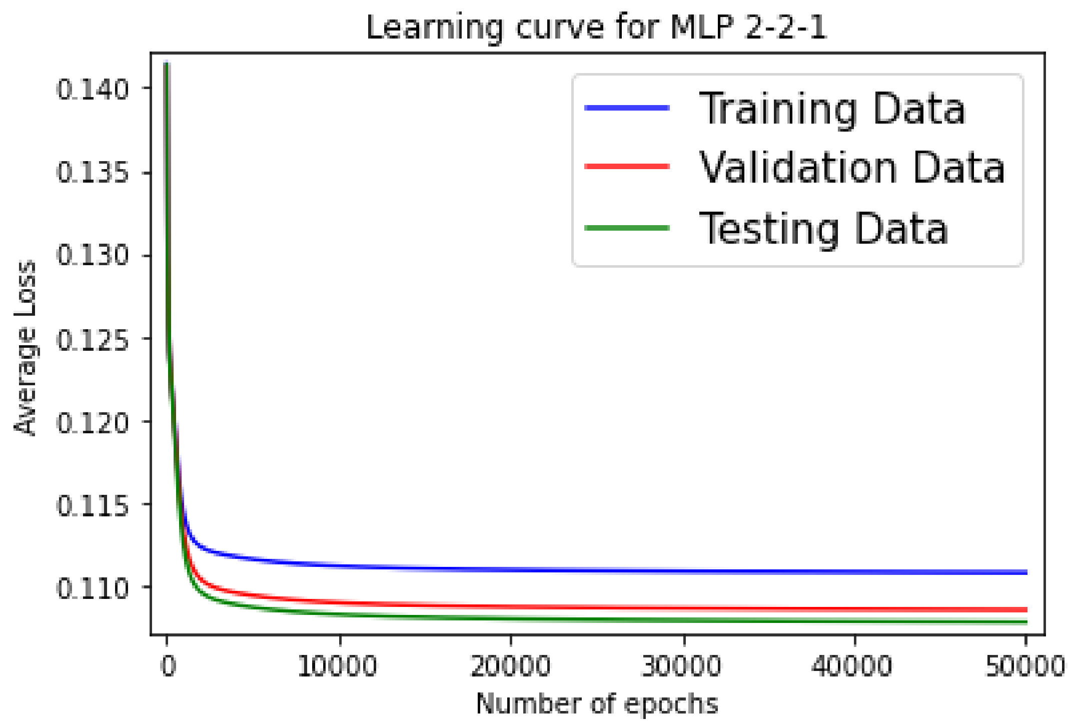
```python
#Normalization of training, validation and test data
mean_vector=np.mean(input_features_train, axis = 0)
std_vector=np.std(input_features_train, axis = 0)

input_features_train=(input_features_train-mean_vector)/std_vector
input_features_val=(input_features_val-mean_vector)/std_vector
input_features_test=(input_features_test-mean_vector)/std_vector
```

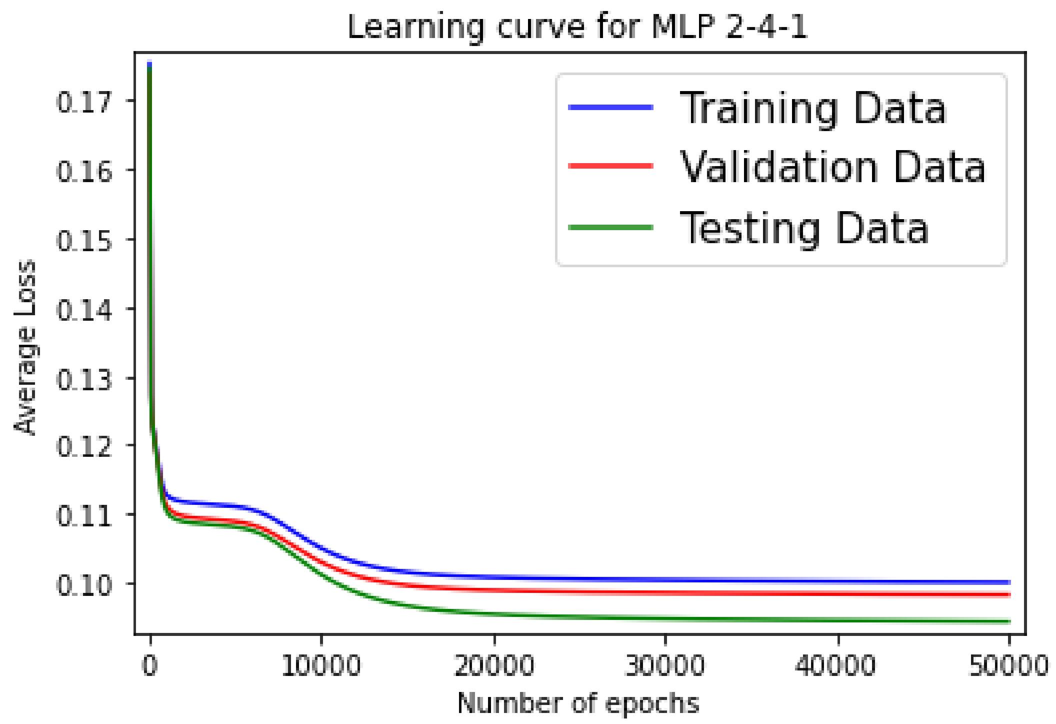
- Output of the main.py file is as below:

  The network is run for the following number of hidden nodes:2,4,6,8,10. For each value of the number of hidden nodes, the network is run multiple times (10 iterations as displayed below) and the accuracy is calculated. The final classification accuracy is thus the average accuracy of all the 10 runs/iterations of the network for the given value of number of hidden nodes.

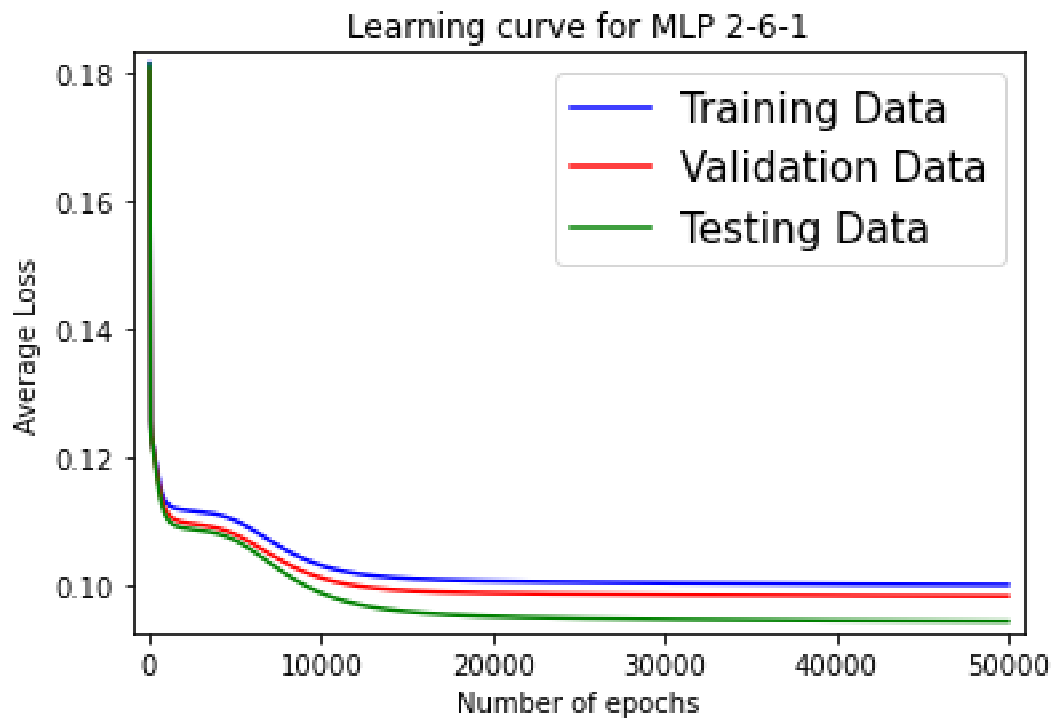
```
Accuracy at number of hidden nodes = 2 for 10 iterations are as below:
Accuracy at Iteration 0 = 0.615
Accuracy at Iteration 1 = 0.615
Accuracy at Iteration 2 = 0.615
Accuracy at Iteration 3 = 0.615
Accuracy at Iteration 4 = 0.615
Accuracy at Iteration 5 = 0.615
Accuracy at Iteration 6 = 0.615
Accuracy at Iteration 7 = 0.615
Accuracy at Iteration 8 = 0.615
Accuracy at Iteration 9 = 0.615
Average Accuracy at number of hidden nodes = 2 is 0.6150000000000001
```



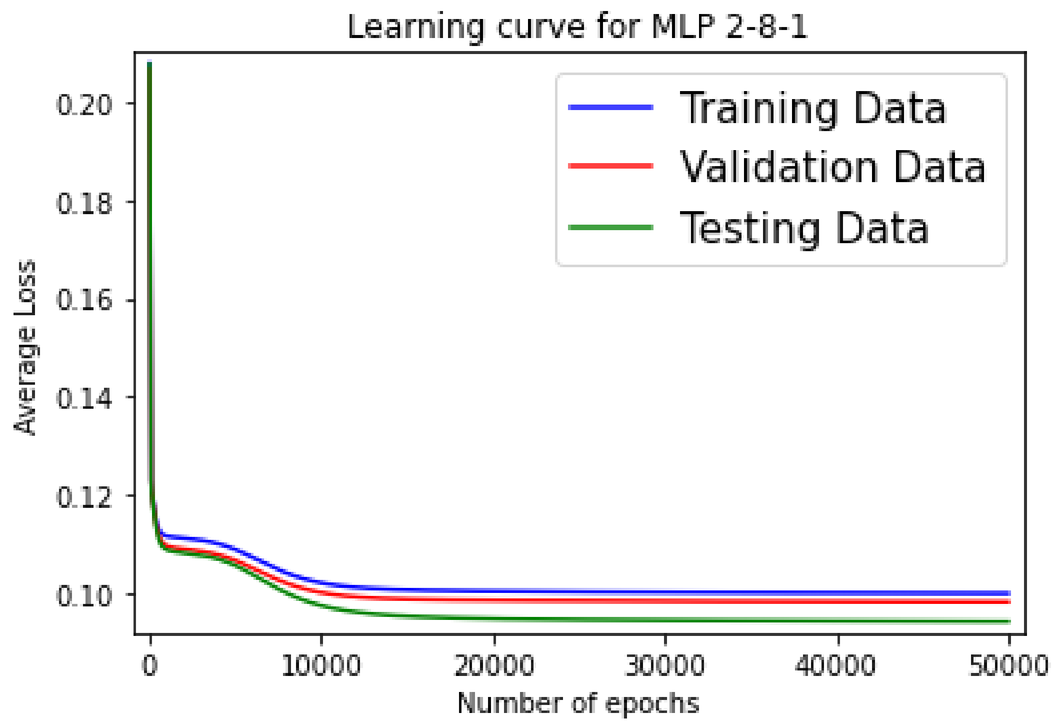
```
Accuracy at number of hidden nodes = 4 for 10 iterations are as below:
Accuracy at Iteration 0 = 0.698
Accuracy at Iteration 1 = 0.698
Accuracy at Iteration 2 = 0.6985
Accuracy at Iteration 3 = 0.698
Accuracy at Iteration 4 = 0.615
Accuracy at Iteration 5 = 0.615
Accuracy at Iteration 6 = 0.697
Accuracy at Iteration 7 = 0.615
Accuracy at Iteration 8 = 0.7785
Accuracy at Iteration 9 = 0.698
Average Accuracy at number of hidden nodes = 4 is 0.6811
```



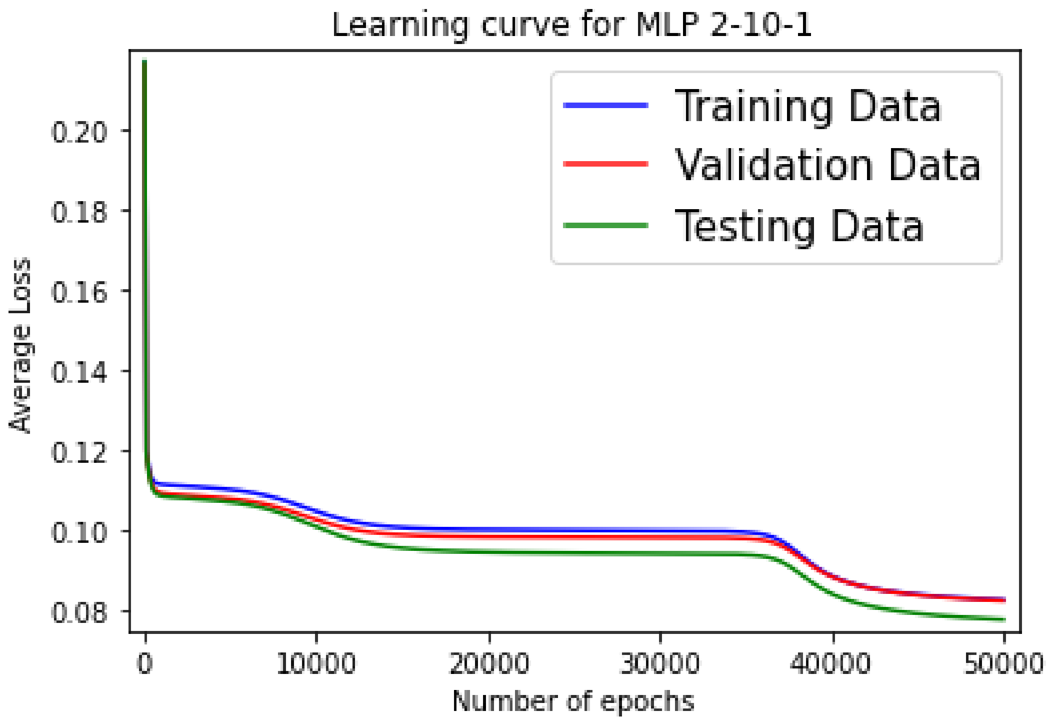
```
Accuracy at number of hidden nodes = 6 for 10 iterations are as below:
Accuracy at Iteration 0 = 0.7775
Accuracy at Iteration 1 = 0.7
Accuracy at Iteration 2 = 0.7785
Accuracy at Iteration 3 = 0.7005
Accuracy at Iteration 4 = 0.701
Accuracy at Iteration 5 = 0.7795
Accuracy at Iteration 6 = 0.615
Accuracy at Iteration 7 = 0.7785
Accuracy at Iteration 8 = 0.7805
Accuracy at Iteration 9 = 0.6995
Average Accuracy at number of hidden nodes = 6 is 0.7310500000000001
```



```
Accuracy at number of hidden nodes = 8 for 10 iterations are as below:
Accuracy at Iteration 0 = 0.78
Accuracy at Iteration 1 = 0.7805
Accuracy at Iteration 2 = 0.78
Accuracy at Iteration 3 = 0.701
Accuracy at Iteration 4 = 0.701
Accuracy at Iteration 5 = 0.6995
Accuracy at Iteration 6 = 0.6995
Accuracy at Iteration 7 = 0.701
Accuracy at Iteration 8 = 0.7015
Accuracy at Iteration 9 = 0.6995
Average Accuracy at number of hidden nodes = 8 is 0.72435
```



```
Accuracy at number of hidden nodes = 10 for 10 iterations are as below:
Accuracy at Iteration 0 = 0.7805
Accuracy at Iteration 1 = 0.7015
Accuracy at Iteration 2 = 0.7795
Accuracy at Iteration 3 = 0.702
Accuracy at Iteration 4 = 0.699
Accuracy at Iteration 5 = 0.768
Accuracy at Iteration 6 = 0.6995
Accuracy at Iteration 7 = 0.78
Accuracy at Iteration 8 = 0.7805
Accuracy at Iteration 9 = 0.786
Average Accuracy at number of hidden nodes = 10 is 0.7476499999999999
```



From the above output of the MLP code (main.py file), we observe that:

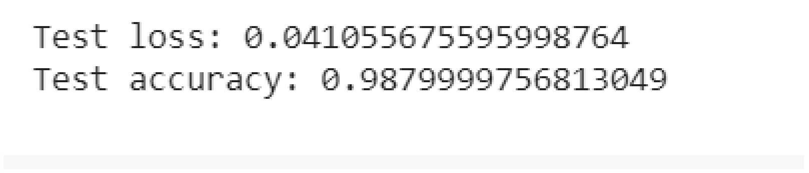
Average Accuracy at number of hidden nodes = 2 is 0.6150000000000001
Average Accuracy at number of hidden nodes = 4 is 0.6811
Average Accuracy at number of hidden nodes = 6 is 0.7310500000000001
Average Accuracy at number of hidden nodes = 8 is 0.72435
Average Accuracy at number of hidden nodes = 10 is **0.7476499999999999 (Best value)**

As we see from the output, the best classification accuracy for the testing set is achieved whe n number of hidden nodes are 10, which is 0.7476499999999999.

Task 2 results

We tried changing various hyper-parameters to verify our hypothesis about them.
The below code snippets can be run from the file **Project2_Task2 .ipynb** to see the output.

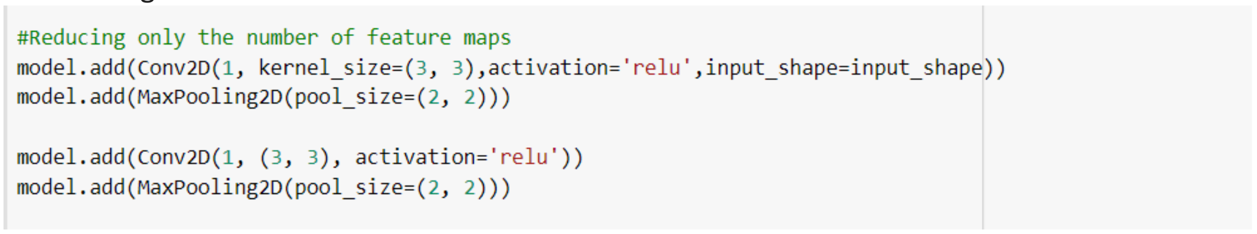
- **BASELINE EXPERIMENT AS SUCH WITHOUT ANY CHANGE**



```
Test loss: 0.041055675595998764
Test accuracy: 0.9879999756813049
```

- **Hypothesis 1**: **Decreasing number of Feature maps**
  Decreasing feature map to 1. The hypothesis is to check that it takes more than 1 convolution feature map to learn generalization on the given data.

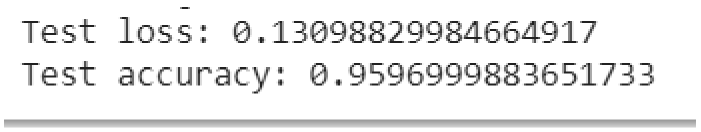
  Code change:



```
#Reducing only the number of feature maps
model.add(Conv2D(1, kernel_size=(3, 3),activation='relu',input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(1, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

  Results:

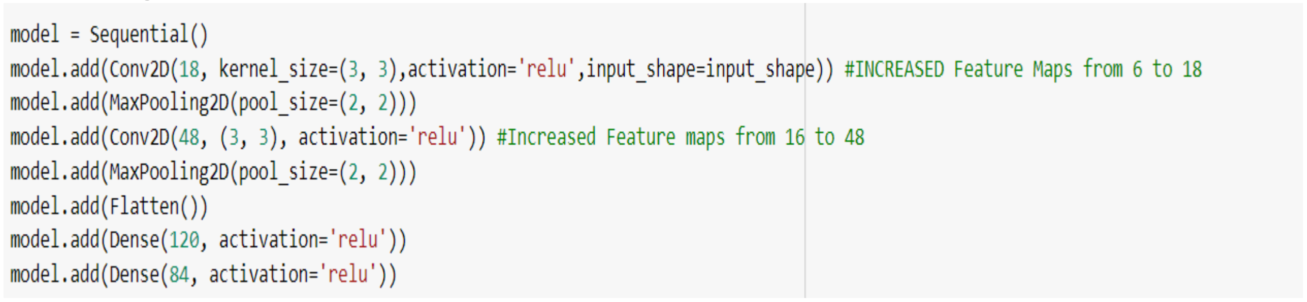


```
Test loss: 0.13098829984664917
Test accuracy: 0.9596999883651733
```

  Remarks:
  Accuracy decreases, which means that our hypothesis was correct that only one convolution features is not enough to represent the data.

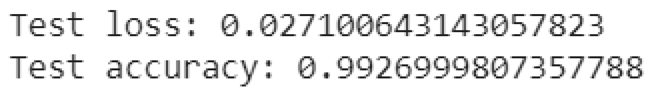
- **Hypothesis 2 : Increasing number of Feature Maps**
  Increasing feature maps by 3 times. The hypothesis is to check that given that model can learn more number of features in convolution, can it learn features that are more relev ant.

Code change:



```
model = Sequential()
model.add(Conv2D(18, kernel_size=(3, 3),activation='relu',input_shape=input_shape)) #INCREASED Feature Maps from 6 to 18
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(48, (3, 3), activation='relu')) #Increased Feature maps from 16 to 48
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(120, activation='relu'))
model.add(Dense(84, activation='relu'))
```

Results:

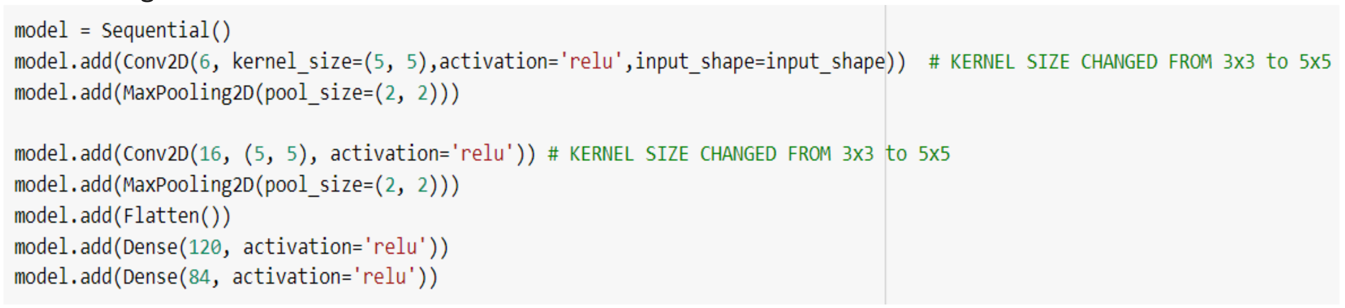


```
Test loss: 0.027100643143057823
Test accuracy: 0.9926999807357788
```

Remarks:
Hypothesis comes out to be true, that is, increasing number of feature maps may result in learning more relevant features, as accuracy increased from baseline accuracy.

- **Hypothesis 3: Kernel Size**
  We will only increase kernel size in this hypothesis, the idea behind this hypothesis is that our images in MNIST data are less complex, so it is possible that we can capture low level features better in a kernel of bigger size

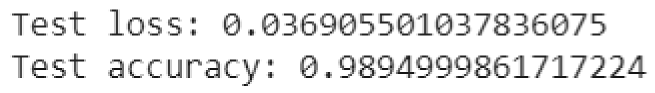
Code change:



```
model = Sequential()
model.add(Conv2D(6, kernel_size=(5, 5),activation='relu',input_shape=input_shape))  # KERNEL SIZE CHANGED FROM 3x3 to 5x5
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(16, (5, 5), activation='relu')) # KERNEL SIZE CHANGED FROM 3x3 to 5x5
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(120, activation='relu'))
model.add(Dense(84, activation='relu'))
```

Results:

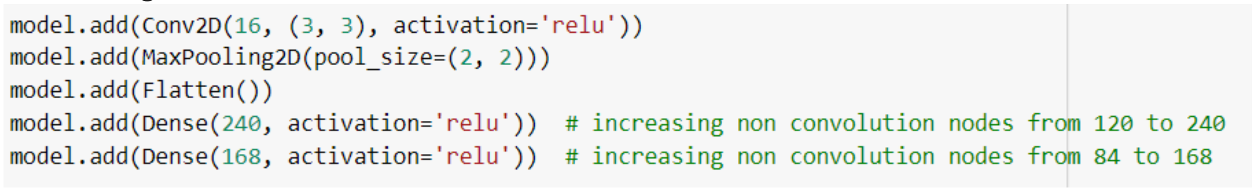


```
Test loss: 0.03690550103783607 5
Test accuracy: 0.9894999861717224
```

Remarks:
We observe slight improvement over our baseline accuracy. This shows the 5x5 is a better choice in our dataset rather than widely used 3x3 , as our low level features are more extractable when we compare a 25 pixel at a time rather than 9.

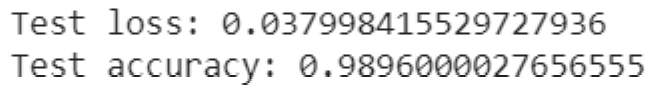
- **Hypothesis 4: Increasing nodes in non convolutional layers**
  In this hypothesis we want to increase nodes in last two layers which are non-convolutional layers. The reason is that more nodes can learn more complex representations, so there might be a better representation possible with more nodes, which can help in learning a better model.

  Code change:



```
model.add(Conv2D(16, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(240, activation='relu'))  # increasing non convolution nodes from 120 to 240
model.add(Dense(168, activation='relu'))  # increasing non convolution nodes from 84 to 168
```

```
Results:
```

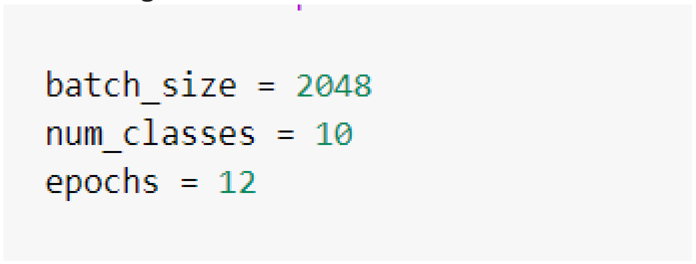


```
Test loss: 0.037998415529727936
Test accuracy: 0.9896000027656555
```

  Remarks:
  We observe slight improvement with respect to our baseline accuracy, so increasing complexity might be helping in learning a better representation.

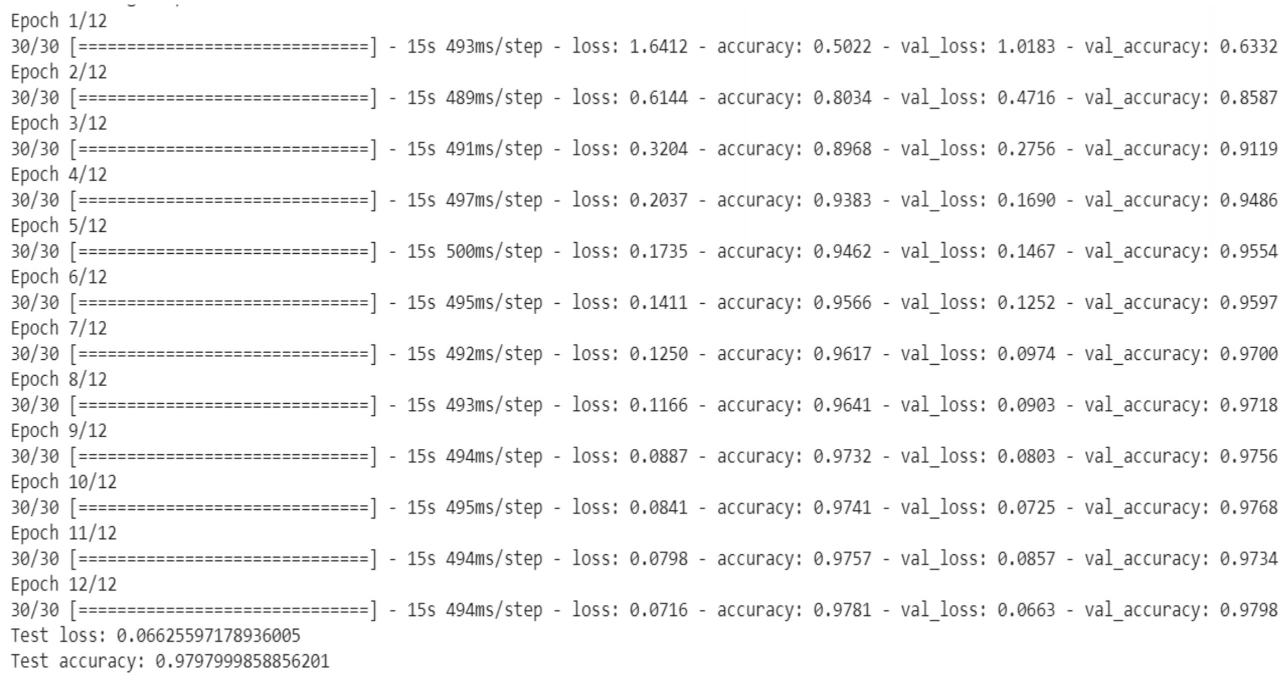
- **Hypothesis 5: Batch Size**
  Larger batch size allows computational speed using parallelization of GPUs. In general, too large of a batch size leads to poor generalization. For convex functions, using a batch equal to the entire dataset guarantees convergence to the global optima of the objective function but this costs a slower convergence to that optima. Using smaller batch have faster convergence to good solutions but not necessarily will reach to global optima. In this hypothesis we will test, if we increase batch size, keeping the epochs same, does the accuracy decrease, as it needs more epoch to generalize.

  Code change:



```
batch_size = 2048
num_classes = 10
epochs = 12
```

Results:

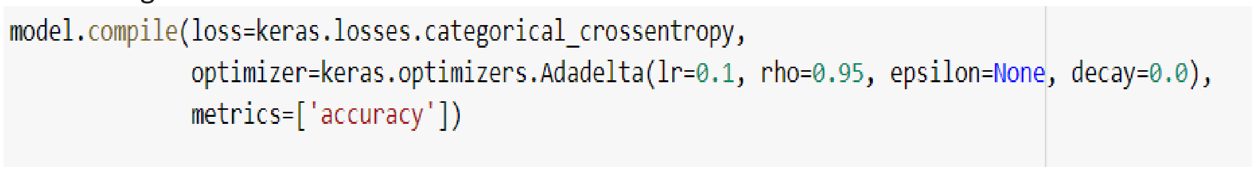


```
Epoch 1/12
30/30 [==============================] - 15s 493ms/step - loss: 1.6412 - accuracy: 0.5022 - val_loss: 1.0183 - val_accuracy: 0.6332
Epoch 2/12
30/30 [==============================] - 15s 489ms/step - loss: 0.6144 - accuracy: 0.8034 - val_loss: 0.4716 - val_accuracy: 0.8587
Epoch 3/12
30/30 [==============================] - 15s 491ms/step - loss: 0.3204 - accuracy: 0.8968 - val_loss: 0.2756 - val_accuracy: 0.9119
Epoch 4/12
30/30 [==============================] - 15s 497ms/step - loss: 0.2037 - accuracy: 0.9383 - val_loss: 0.1690 - val_accuracy: 0.9486
Epoch 5/12
30/30 [==============================] - 15s 500ms/step - loss: 0.1735 - accuracy: 0.9462 - val_loss: 0.1467 - val_accuracy: 0.9554
Epoch 6/12
30/30 [==============================] - 15s 495ms/step - loss: 0.1411 - accuracy: 0.9566 - val_loss: 0.1252 - val_accuracy: 0.9597
Epoch 7/12
30/30 [==============================] - 15s 492ms/step - loss: 0.1250 - accuracy: 0.9617 - val_loss: 0.0974 - val_accuracy: 0.9700
Epoch 8/12
30/30 [==============================] - 15s 493ms/step - loss: 0.1166 - accuracy: 0.9641 - val_loss: 0.0903 - val_accuracy: 0.9718
Epoch 9/12
30/30 [==============================] - 15s 494ms/step - loss: 0.0887 - accuracy: 0.9732 - val_loss: 0.0803 - val_accuracy: 0.9756
Epoch 10/12
30/30 [==============================] - 15s 495ms/step - loss: 0.0841 - accuracy: 0.9741 - val_loss: 0.0725 - val_accuracy: 0.9768
Epoch 11/12
30/30 [==============================] - 15s 494ms/step - loss: 0.0798 - accuracy: 0.9757 - val_loss: 0.0857 - val_accuracy: 0.9734
Epoch 12/12
30/30 [==============================] - 15s 494ms/step - loss: 0.0716 - accuracy: 0.9781 - val_loss: 0.0663 - val_accuracy: 0.9798
Test loss: 0.06625597178936005
Test accuracy: 0.9797999858856201
```

Remarks:
Since Validation acc > training acc, we can run it for more epochs, this shows that larger batch size takes mores epochs to generalize.

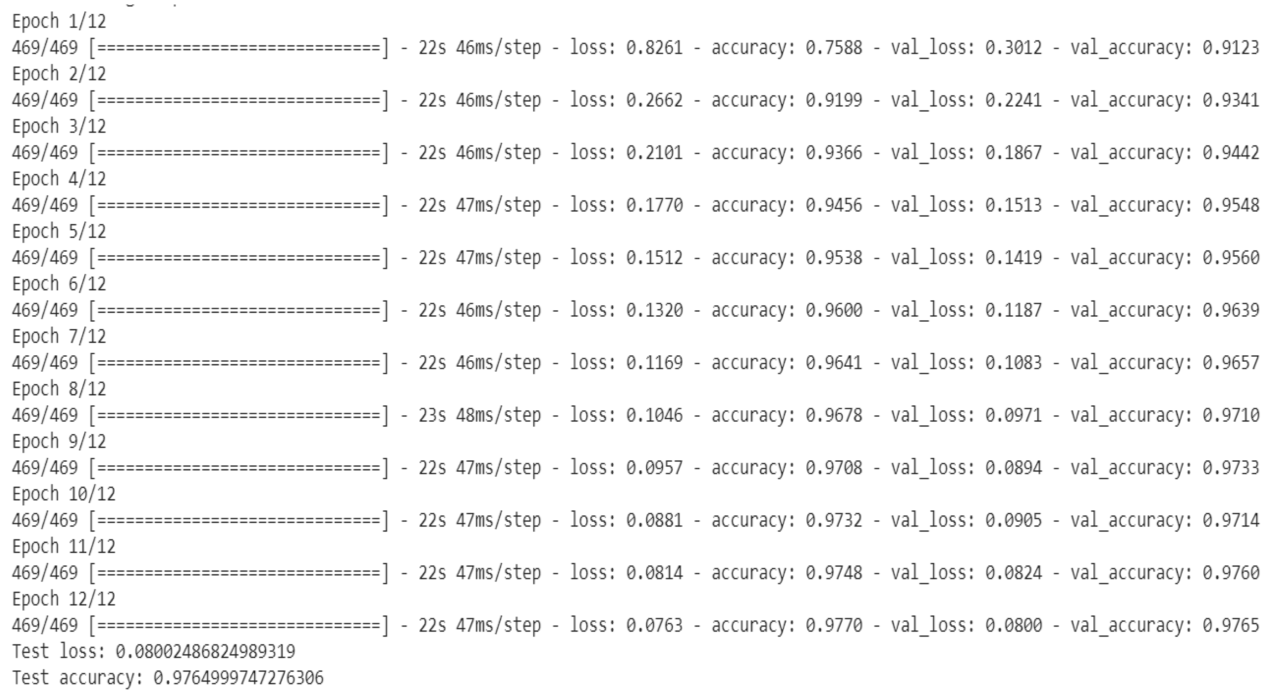
- **Hypothesis 6: Decreasing Learning rate**
  Ideal learning rate is hard to predict, a slower learninig rate can increase compuational complexity whereas a large one can take us away from global optima. In this hypothesis, we will test with slower learning rate and show that in same number of epochs it will not converge, in other words it needs more epochs to converge.

  Code change:



```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(lr=0.1, rho=0.95, epsilon=None, decay=0.0),
              metrics=['accuracy'])
```

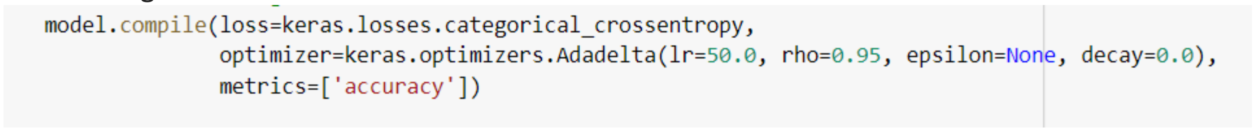
Results:



```
Epoch 1/12
469/469 [==============================] - 22s 46ms/step - loss: 0.8261 - accuracy: 0.7588 - val_loss: 0.3012 - val_accuracy: 0.9123
Epoch 2/12
469/469 [==============================] - 22s 46ms/step - loss: 0.2662 - accuracy: 0.9199 - val_loss: 0.2241 - val_accuracy: 0.9341
Epoch 3/12
469/469 [==============================] - 22s 46ms/step - loss: 0.2101 - accuracy: 0.9366 - val_loss: 0.1867 - val_accuracy: 0.9442
Epoch 4/12
469/469 [==============================] - 22s 47ms/step - loss: 0.1770 - accuracy: 0.9456 - val_loss: 0.1513 - val_accuracy: 0.9548
Epoch 5/12
469/469 [==============================] - 22s 47ms/step - loss: 0.1512 - accuracy: 0.9538 - val_loss: 0.1419 - val_accuracy: 0.9560
Epoch 6/12
469/469 [==============================] - 22s 46ms/step - loss: 0.1320 - accuracy: 0.9600 - val_loss: 0.1187 - val_accuracy: 0.9639
Epoch 7/12
469/469 [==============================] - 22s 46ms/step - loss: 0.1169 - accuracy: 0.9641 - val_loss: 0.1083 - val_accuracy: 0.9657
Epoch 8/12
469/469 [==============================] - 23s 48ms/step - loss: 0.1046 - accuracy: 0.9678 - val_loss: 0.0971 - val_accuracy: 0.9710
Epoch 9/12
469/469 [==============================] - 22s 47ms/step - loss: 0.0957 - accuracy: 0.9708 - val_loss: 0.0894 - val_accuracy: 0.9733
Epoch 10/12
469/469 [==============================] - 22s 47ms/step - loss: 0.0881 - accuracy: 0.9732 - val_loss: 0.0905 - val_accuracy: 0.9714
Epoch 11/12
469/469 [==============================] - 22s 47ms/step - loss: 0.0814 - accuracy: 0.9748 - val_loss: 0.0824 - val_accuracy: 0.9760
Epoch 12/12
469/469 [==============================] - 22s 47ms/step - loss: 0.0763 - accuracy: 0.9770 - val_loss: 0.0800 - val_accuracy: 0.9765
Test loss: 0.08002486824989319
Test accuracy: 0.9764999747276306
```

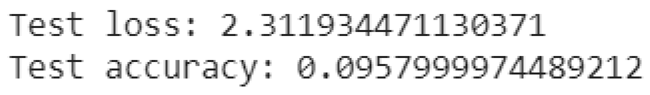
- **Hypothesis 7: Increasing Learning rate**
  Ideal learning rate is hard to predict, a slower learning rate can increase computational complexity whereas a large one can take us away from the global optima. In this hypothesis, we will test with higher learning rate and show that it is taking us away from the solution.

  Code change:



```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(lr=50.0, rho=0.95, epsilon=None, decay=0.0),
              metrics=['accuracy'])
```

  Results:



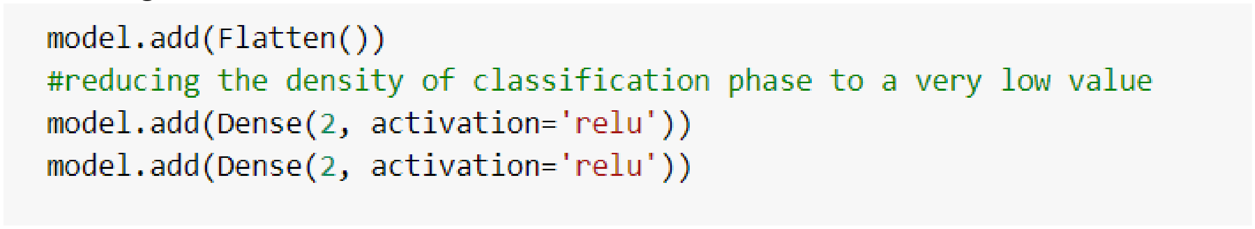
```
Test loss: 2.311934471130371
Test accuracy: 0.0957999974489212
```

  Remarks:

So far this is the least accuracy we have seen, higher learning rate is shifting us away from target vector.

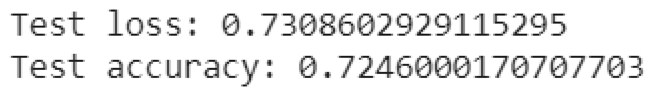
- **Hypothesis 8: Decreasing nodes in non convolutional layers**
  In this hypothesis we want to decrease nodes in last two layers which are non-convolutional layers. The reason is that we want to check if convolutional layers itself can act as final features, without representing them again in higher dimension before the output layer.

  Code change:



```
model.add(Flatten())
#reducing the density of classification phase to a very low value
model.add(Dense(2, activation='relu'))
model.add(Dense(2, activation='relu'))
```

  Results:

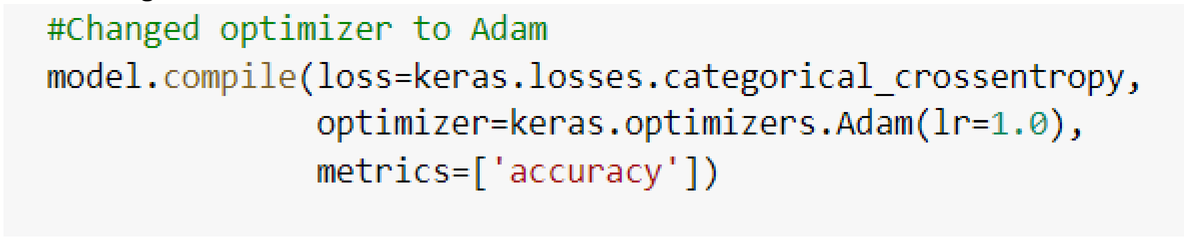


```
Test loss: 0.7308602929115295
Test accuracy: 0.7246000170707703
```

  Remarks:
  This shows that representiting the output from convolutional layers in higher dimiension is important to learn and use features from it, to train the model.

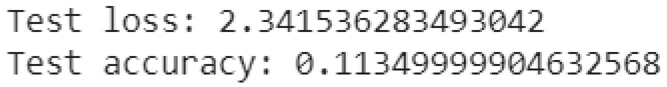
- **Hypothesis 9: Changing optimizer to Adam**
  There are different types of gradient descent optimizers. In baseline code, we have used adadelta, which adapts learning rates based on a moving window of gradient updates. I n this hypothesis we want to test Adam, which is based on the adaptive estimation of fir st-order and second-order moments.

  Code change:



```
#Changed optimizer to Adam
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(lr=1.0),
              metrics=['accuracy'])
```

Results:



```
Test loss: 2.341536283493042
Test accuracy: 0.11349999904632568
```

Remarks:

Hence, this optimizer is not working in this scenario.