Recognition Using Deep Networks

Malhar Mahant and Kruthika Gangaraju

Description:

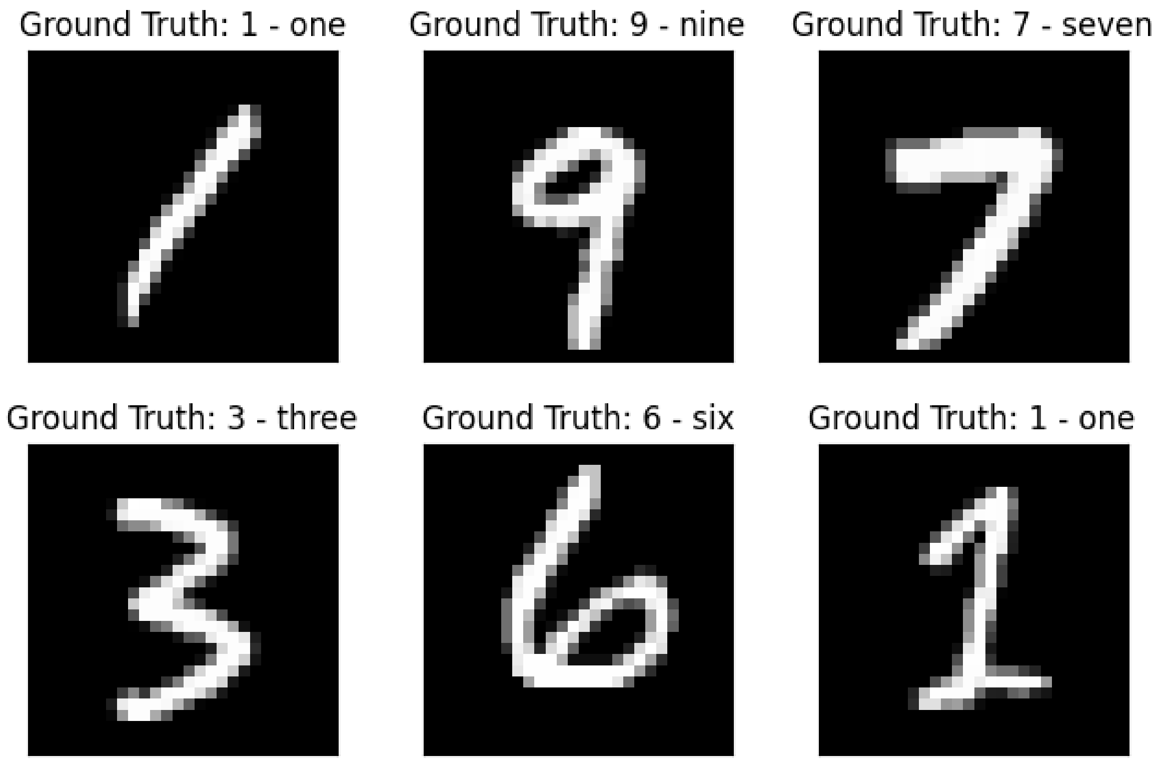
In this project we trained and tested deep network models. We used the MNIST dataset for recognition. For this project, we built and developed a MNIST data set, plotted the training and testing accuracy and tested the same model for handwritten numbers. The project helped demonstrate the capabilities that deep networks possess in accomplishing computer vision tasks.

Please refer to the readme for instructions on how to use the application.
Time Travel Days used for this project: 2.

Task 1:

For the first task, we directly imported the MNIST dataset which consists of 60,000 28x28 labeled training set and 10,000 28x28 labeled testing set.

A. Get the MNIST digit data set: We loaded the MNIST dataset. Using matplotlib, we plotted the first six example digits.



B. Make your code repeatable: We made the code repeatable by turning off CUDA. Turning off CUDA for repeatability means running the code on the CPU instead of the GPU. By running the code on the CPU, we can eliminate these sources of variability and ensure that the results are consistent and reproducible.
C. Build a network model: Once the dataset is loaded, we built a network model using two convolution layers (10 5x5 filters and 20 5x5 filters), two pooling layers (2x2 window with a ReLu function applied) and a dropout layer.

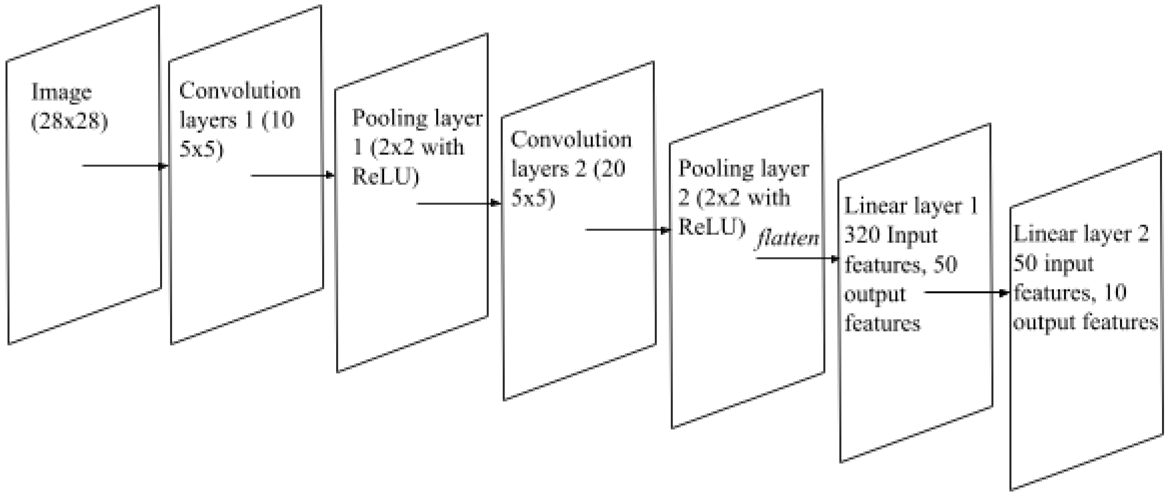
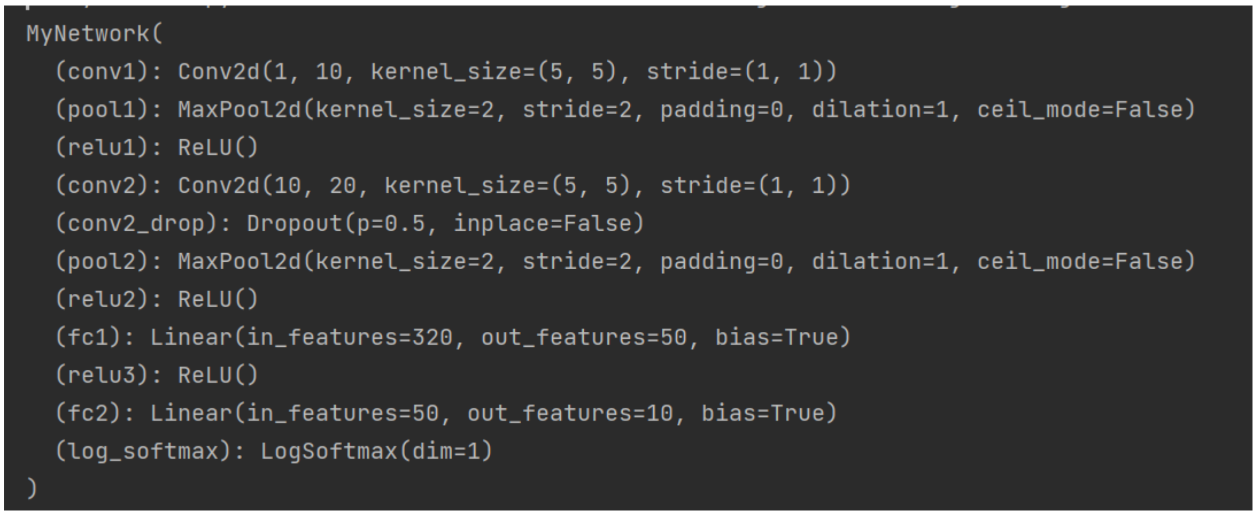
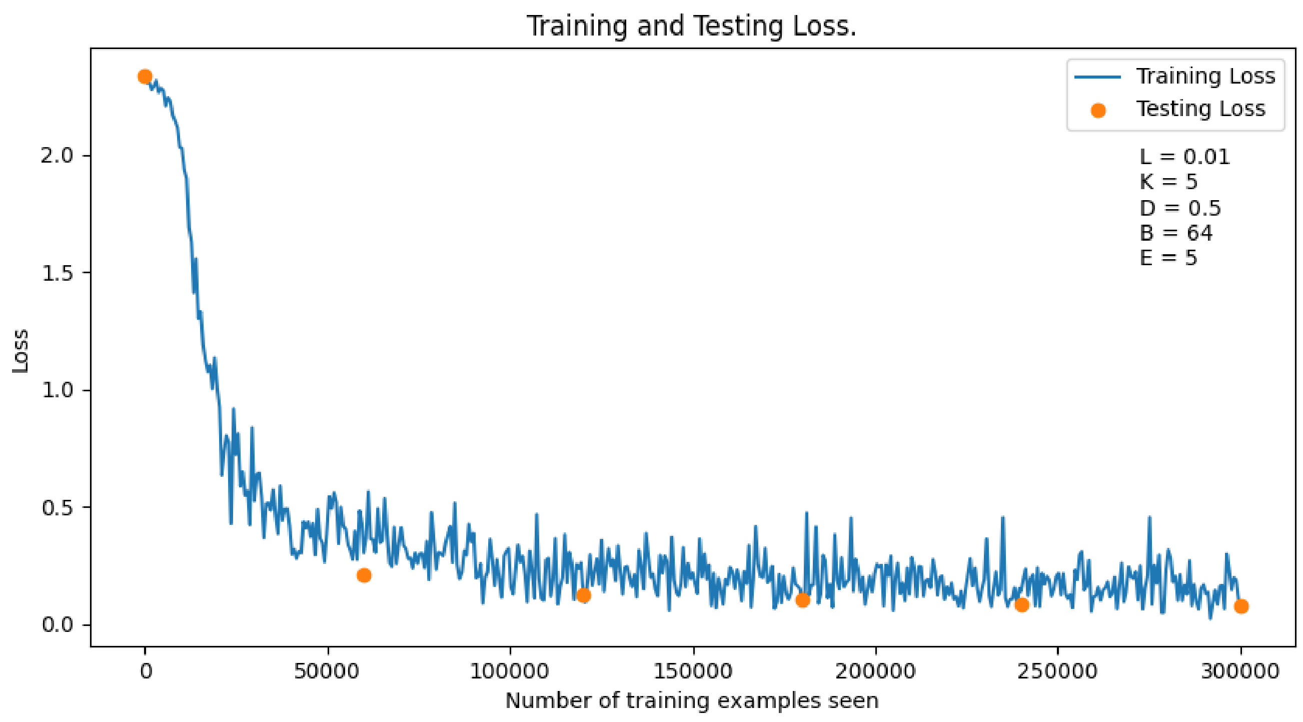


Image (28x28)
Convolution layers 1 (10 5x5)
Pooling layer 1 (2x2 with ReLU)
Convolution layers 2 (20 5x5)
Pooling layer 2 (2x2 with ReLU)
flatten
Linear layer 1 320 Input features, 50 output features
Linear layer 2 50 input features, 10 output features

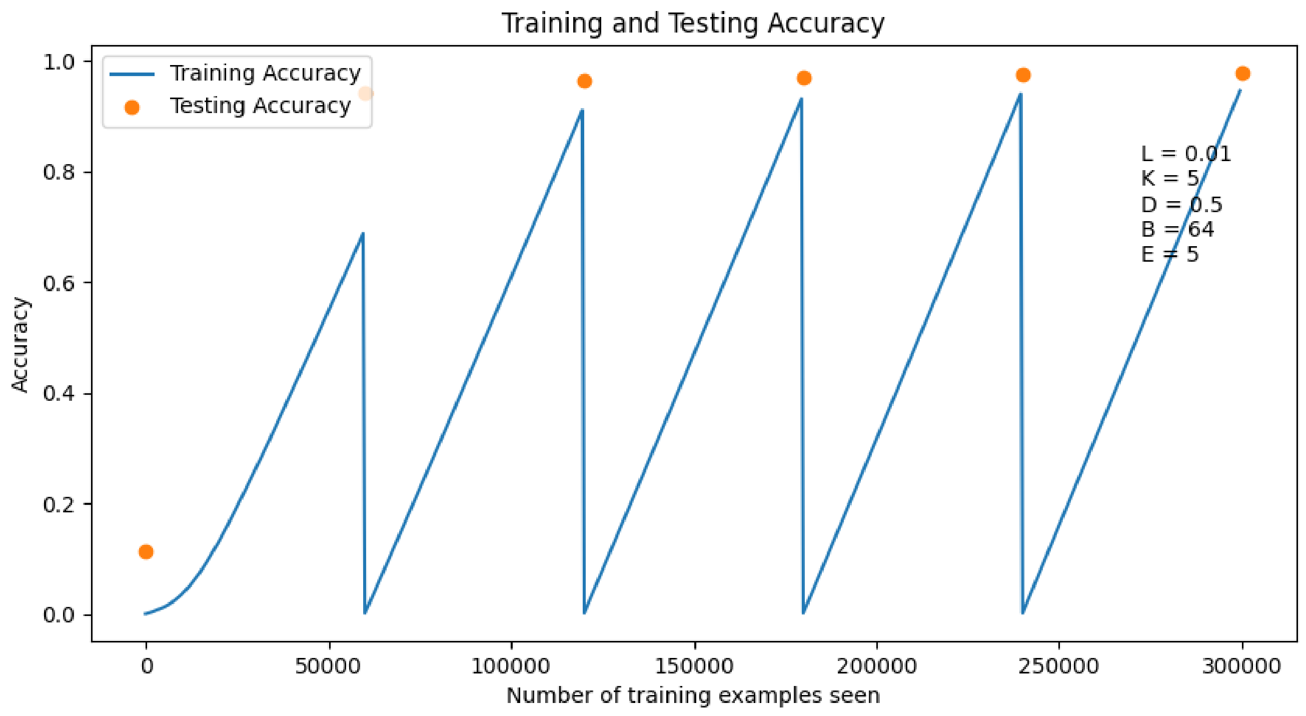
Printing model structure to console.


```
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu1): ReLU()
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout(p=0.5, inplace=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu2): ReLU()
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (log_softmax): LogSoftmax(dim=1)
)
```

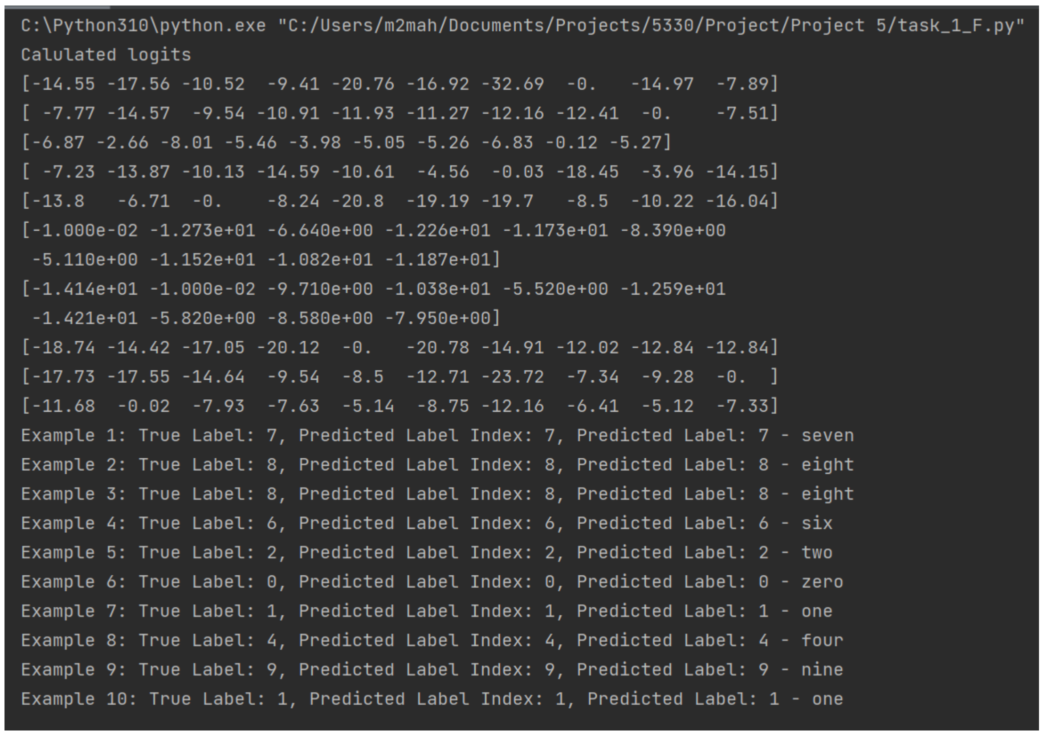
D. Train the model: We trained the model one epoch at a time using 5 epochs. We used a batch size of 64 for training.
Plot of training and test loss:



Plot of training and test accuracy: The ups and downs are plots as the training progresses through each epoch. It starts from the bottom for each epoch.

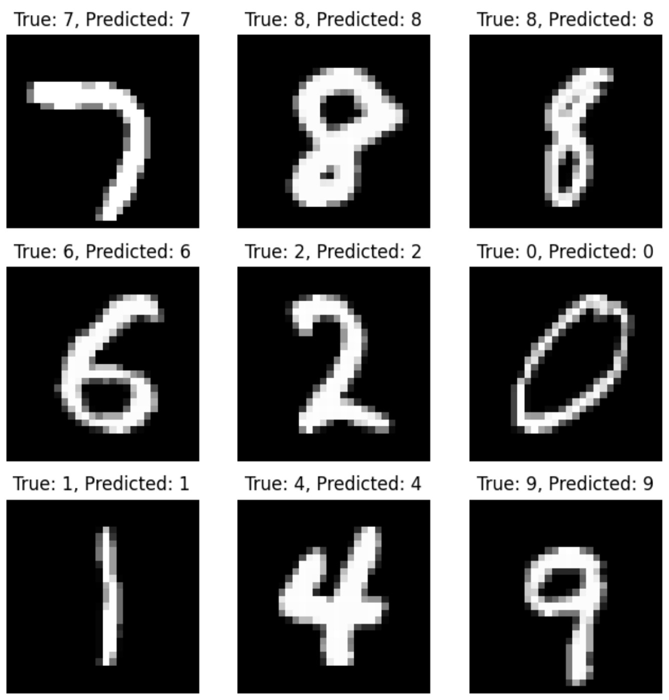


E. Save the model: We saved the model we trained. The model is saved according to the name passed to the train_network function in task_1_A_E.py
F. Read the network and run it on the test set: After training and saving the model, we tested it on a sample. Printed output values, And the true label, predicted label index (calculated from output values) and the predicted label:

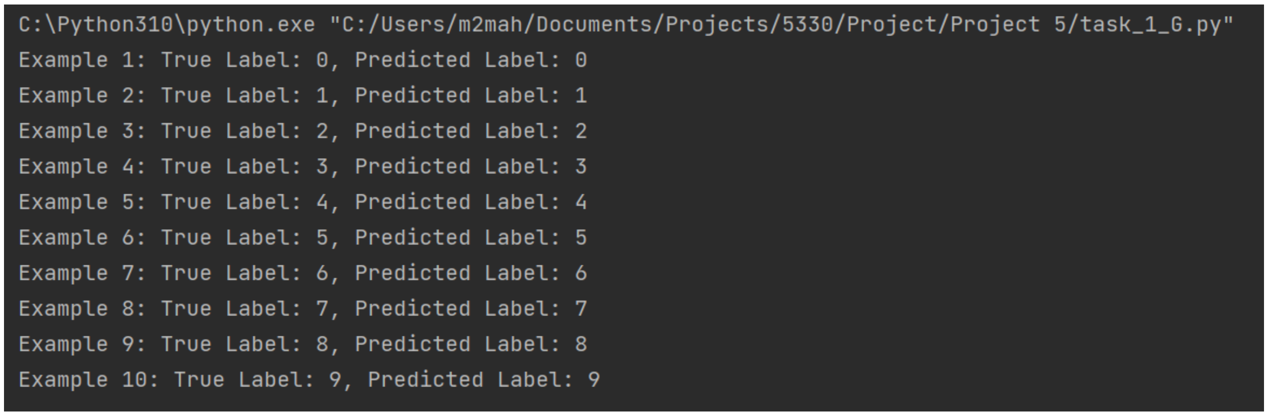


```
C:\Python310\python.exe "C:/Users/m2mah/Documents/Projects/5330/Project/Project 5/task_1_F.py"
Calulated logits
[-14.55 -17.56 -10.52  -9.41 -20.76 -16.92 -32.69  -0.   -14.97  -7.89]
[ -7.77 -14.57  -9.54 -10.91 -11.93 -11.27 -12.16 -12.41  -0.    -7.51]
[-6.87 -2.66 -8.01 -5.46 -3.98 -5.05 -5.26 -6.83 -0.12 -5.27]
[ -7.23 -13.87 -10.13 -14.59 -10.61  -4.56  -0.03 -18.45  -3.96 -14.15]
[-13.8   -6.71  -0.    -8.24 -20.8  -19.19 -19.7   -8.5  -10.22 -16.04]
[-1.000e-02 -1.273e+01 -6.640e+00 -1.226e+01 -1.173e+01 -8.390e+00
 -5.110e+00 -1.152e+01 -1.082e+01 -1.187e+01]
[-1.414e+01 -1.000e-02 -9.710e+00 -1.038e+01 -5.520e+00 -1.259e+01
 -1.421e+01 -5.820e+00 -8.580e+00 -7.950e+00]
[-18.74 -14.42 -17.05 -20.12  -0.   -20.78 -14.91 -12.02 -12.84 -12.84]
[-17.73 -17.55 -14.64  -9.54  -8.5  -12.71 -23.72  -7.34  -9.28  -0.  ]
[-11.68  -0.02  -7.93  -7.63  -5.14  -8.75 -12.16  -6.41  -5.12  -7.33]
Example 1: True Label: 7, Predicted Label Index: 7, Predicted Label: 7 - seven
Example 2: True Label: 8, Predicted Label Index: 8, Predicted Label: 8 - eight
Example 3: True Label: 8, Predicted Label Index: 8, Predicted Label: 8 - eight
Example 4: True Label: 6, Predicted Label Index: 6, Predicted Label: 6 - six
Example 5: True Label: 2, Predicted Label Index: 2, Predicted Label: 2 - two
Example 6: True Label: 0, Predicted Label Index: 0, Predicted Label: 0 - zero
Example 7: True Label: 1, Predicted Label Index: 1, Predicted Label: 1 - one
Example 8: True Label: 4, Predicted Label Index: 4, Predicted Label: 4 - four
Example 9: True Label: 9, Predicted Label Index: 9, Predicted Label: 9 - nine
Example 10: True Label: 1, Predicted Label Index: 1, Predicted Label: 1 - one
```

Plot of the first 9 digits:

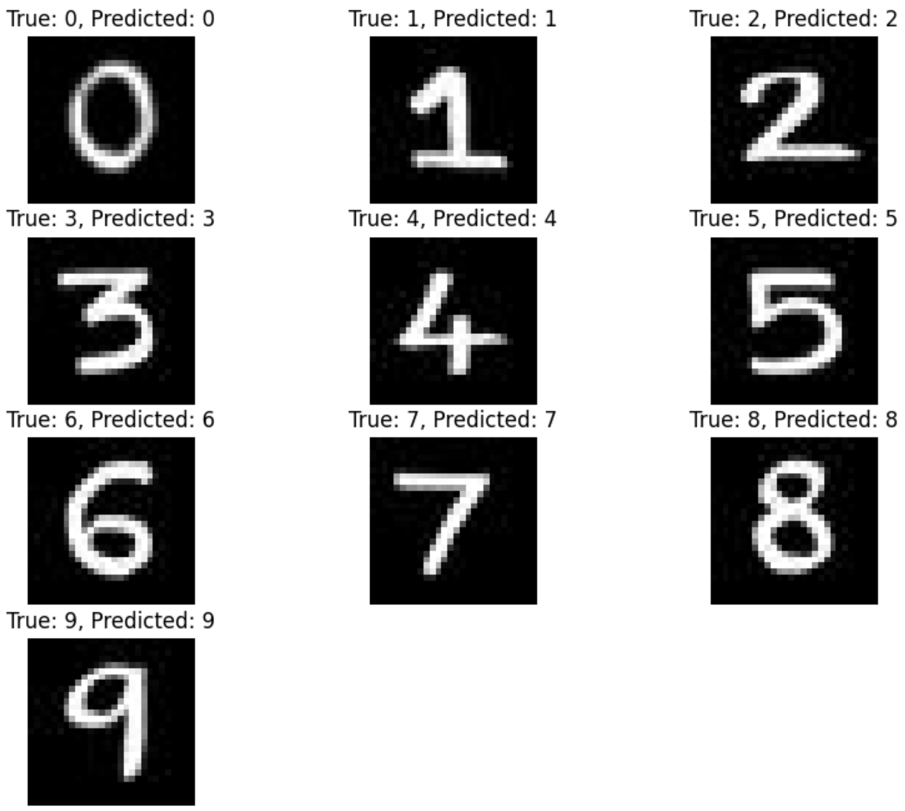


G. Test the network on new inputs: We ran the same model on a sample set of handwritten digits and got the prediction results for the same. The handwritten digit images are resized and matched the intensities to the test model.
   Printing the test results on console: We see a 100% accuracy on the new handwritten digit images. This could be because the handwriting that we tested is very similar to the MNIST dataset.



```
C:\Python310\python.exe "C:/Users/m2mah/Documents/Projects/5330/Project/Project 5/task_1_G.py"
Example 1: True Label: 0, Predicted Label: 0
Example 2: True Label: 1, Predicted Label: 1
Example 3: True Label: 2, Predicted Label: 2
Example 4: True Label: 3, Predicted Label: 3
Example 5: True Label: 4, Predicted Label: 4
Example 6: True Label: 5, Predicted Label: 5
Example 7: True Label: 6, Predicted Label: 6
Example 8: True Label: 7, Predicted Label: 7
Example 9: True Label: 8, Predicted Label: 8
Example 10: True Label: 9, Predicted Label: 9
```

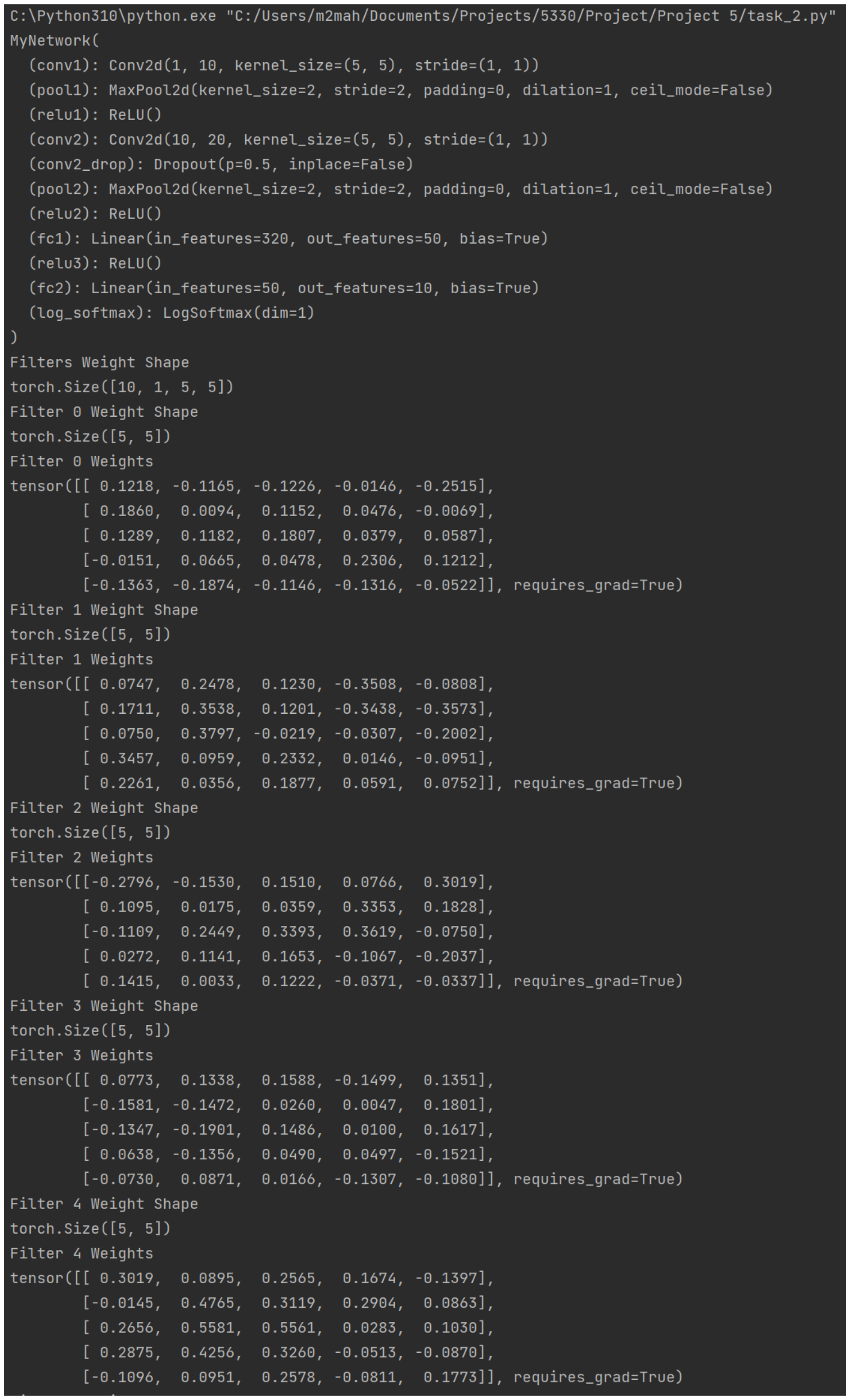
   Plot of the tested images and their predictions and true labels:



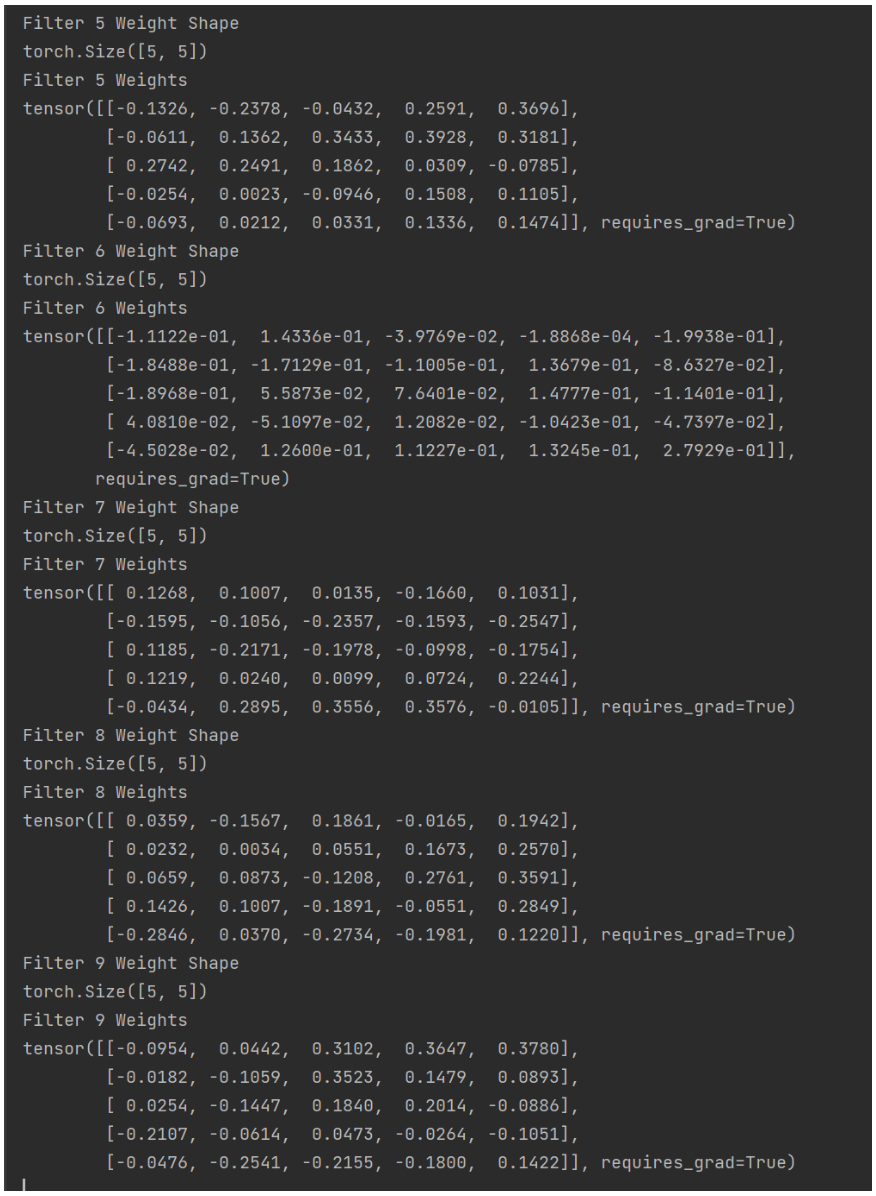
Task 2:

For the second task, we analyzed the network we developed. By doing this we got the structure of our network and name of each layer.

A. Analyze the first layer: We got the weights of the first layer which gives a tensor of the filters in that layer. Using pyplot, we got a plot of the filters.
   Print of weights on console:

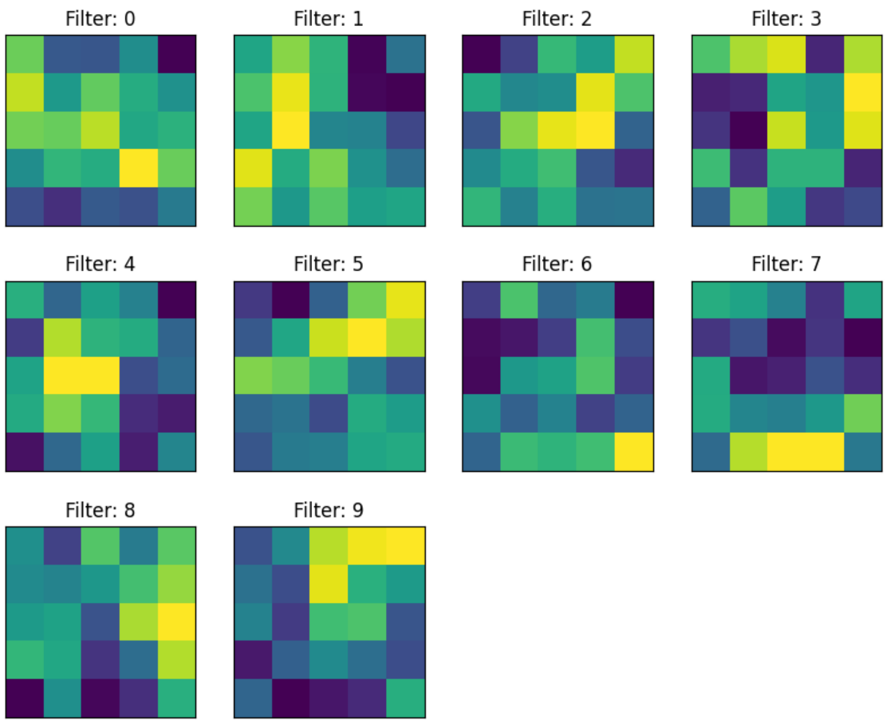


```
C:\Python310\python.exe "C:/Users/m2mah/Documents/Projects/5330/Project/Project 5/task_2.py"
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu1): ReLU()
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout(p=0.5, inplace=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu2): ReLU()
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (log_softmax): LogSoftmax(dim=1)
)
Filters Weight Shape
torch.Size([10, 1, 5, 5])
Filter 0 Weight Shape
torch.Size([5, 5])
Filter 0 Weights
tensor([[ 0.1218, -0.1165, -0.1226, -0.0146, -0.2515],
        [ 0.1860,  0.0094,  0.1152,  0.0476, -0.0069],
        [ 0.1289,  0.1182,  0.1807,  0.0379,  0.0587],
        [-0.0151,  0.0665,  0.0478,  0.2306,  0.1212],
        [-0.1363, -0.1874, -0.1146, -0.1316, -0.0522]], requires_grad=True)
Filter 1 Weight Shape
torch.Size([5, 5])
Filter 1 Weights
tensor([[ 0.0747,  0.2478,  0.1230, -0.3508, -0.0808],
        [ 0.1711,  0.3538,  0.1201, -0.3438, -0.3573],
        [ 0.0750,  0.3797, -0.0219, -0.0307, -0.2002],
        [ 0.3457,  0.0959,  0.2332,  0.0146, -0.0951],
        [ 0.2261,  0.0356,  0.1877,  0.0591,  0.0752]], requires_grad=True)
Filter 2 Weight Shape
torch.Size([5, 5])
Filter 2 Weights
tensor([[-0.2796, -0.1530,  0.1510,  0.0766,  0.3019],
        [ 0.1095,  0.0175,  0.0359,  0.3353,  0.1828],
        [-0.1109,  0.2449,  0.3393,  0.3619, -0.0750],
        [ 0.0272,  0.1141,  0.1653, -0.1067, -0.2037],
        [ 0.1415,  0.0033,  0.1222, -0.0371, -0.0337]], requires_grad=True)
Filter 3 Weight Shape
torch.Size([5, 5])
Filter 3 Weights
tensor([[ 0.0773,  0.1338,  0.1588, -0.1499,  0.1351],
        [-0.1581, -0.1472,  0.0260,  0.0047,  0.1801],
        [-0.1347, -0.1901,  0.1486,  0.0100,  0.1617],
        [ 0.0638, -0.1356,  0.0490,  0.0497, -0.1521],
        [-0.0730,  0.0871,  0.0166, -0.1307, -0.1080]], requires_grad=True)
Filter 4 Weight Shape
torch.Size([5, 5])
Filter 4 Weights
tensor([[ 0.3019,  0.0895,  0.2565,  0.1674, -0.1397],
        [-0.0145,  0.4765,  0.3119,  0.2904,  0.0863],
        [ 0.2656,  0.5581,  0.5561,  0.0283,  0.1030],
        [ 0.2875,  0.4256,  0.3260, -0.0513, -0.0870],
        [-0.1096,  0.0951,  0.2578, -0.0811,  0.1773]], requires_grad=True)
```

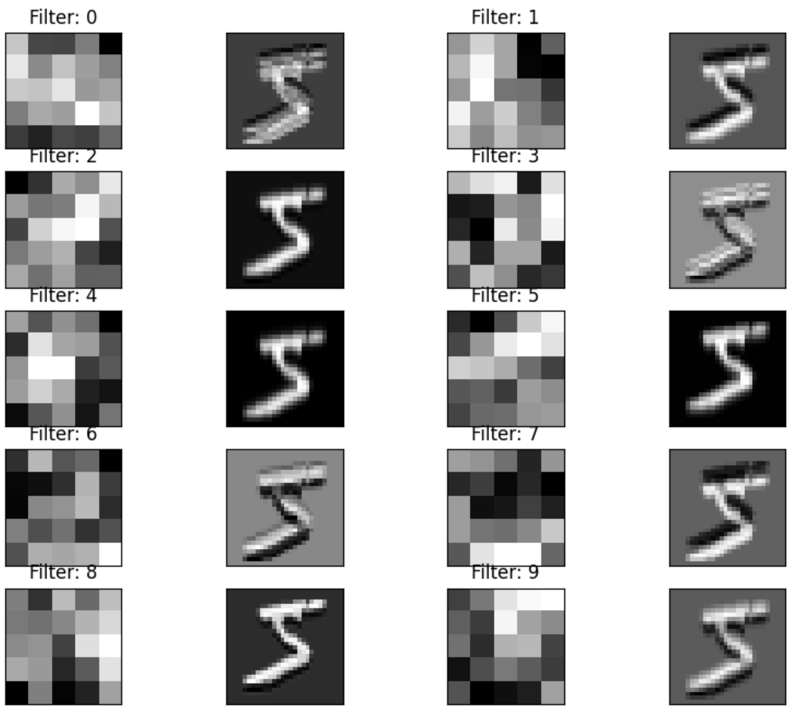


```
Filter 5 Weight Shape
torch.Size([5, 5])
Filter 5 Weights
tensor([[-0.1326, -0.2378, -0.0432,  0.2591,  0.3696],
        [-0.0611,  0.1362,  0.3433,  0.3928,  0.3181],
        [ 0.2742,  0.2491,  0.1862,  0.0309, -0.0785],
        [-0.0254,  0.0023, -0.0946,  0.1508,  0.1105],
        [-0.0693,  0.0212,  0.0331,  0.1336,  0.1474]], requires_grad=True)
Filter 6 Weight Shape
torch.Size([5, 5])
Filter 6 Weights
tensor([[-1.1122e-01,  1.4336e-01, -3.9769e-02, -1.8868e-04, -1.9938e-01],
        [-1.8488e-01, -1.7129e-01, -1.1005e-01,  1.3679e-01, -8.6327e-02],
        [-1.8968e-01,  5.5873e-02,  7.6401e-02,  1.4777e-01, -1.1401e-01],
        [ 4.0810e-02, -5.1097e-02,  1.2082e-02, -1.0423e-01, -4.7397e-02],
        [-4.5028e-02,  1.2600e-01,  1.1227e-01,  1.3245e-01,  2.7929e-01]],
       requires_grad=True)
Filter 7 Weight Shape
torch.Size([5, 5])
Filter 7 Weights
tensor([[ 0.1268,  0.1007,  0.0135, -0.1660,  0.1031],
        [-0.1595, -0.1056, -0.2357, -0.1593, -0.2547],
        [ 0.1185, -0.2171, -0.1978, -0.0998, -0.1754],
        [ 0.1219,  0.0240,  0.0099,  0.0724,  0.2244],
        [-0.0434,  0.2895,  0.3556,  0.3576, -0.0105]], requires_grad=True)
Filter 8 Weight Shape
torch.Size([5, 5])
Filter 8 Weights
tensor([[ 0.0359, -0.1567,  0.1861, -0.0165,  0.1942],
        [ 0.0232,  0.0034,  0.0551,  0.1673,  0.2570],
        [ 0.0659,  0.0873, -0.1208,  0.2761,  0.3591],
        [ 0.1426,  0.1007, -0.1891, -0.0551,  0.2849],
        [-0.2846,  0.0370, -0.2734, -0.1981,  0.1220]], requires_grad=True)
Filter 9 Weight Shape
torch.Size([5, 5])
Filter 9 Weights
tensor([[-0.0954,  0.0442,  0.3102,  0.3647,  0.3780],
        [-0.0182, -0.1059,  0.3523,  0.1479,  0.0893],
        [ 0.0254, -0.1447,  0.1840,  0.2014, -0.0886],
        [-0.2107, -0.0614,  0.0473, -0.0264, -0.1051],
        [-0.0476, -0.2541, -0.2155, -0.1800,  0.1422]], requires_grad=True)
```

Visualization of the 10 filters:



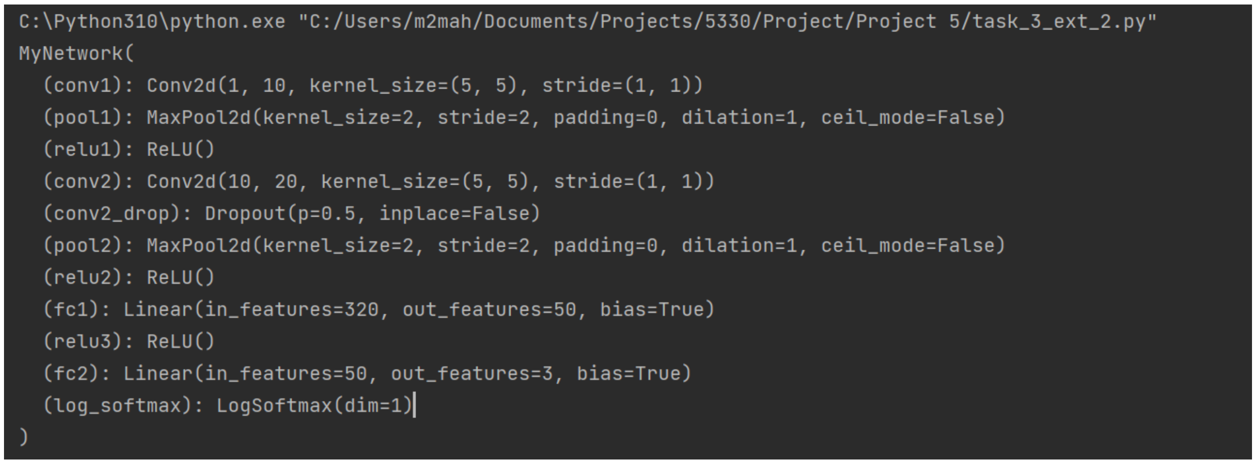
B. Show the effect of filters: We applied the filters to the first training example image using OpenCV's filter2D function. We plot the filters and the result of applying the filter to the image using pyplot.



The resulting images after applying the filters seem consistent given the weights of the filters.

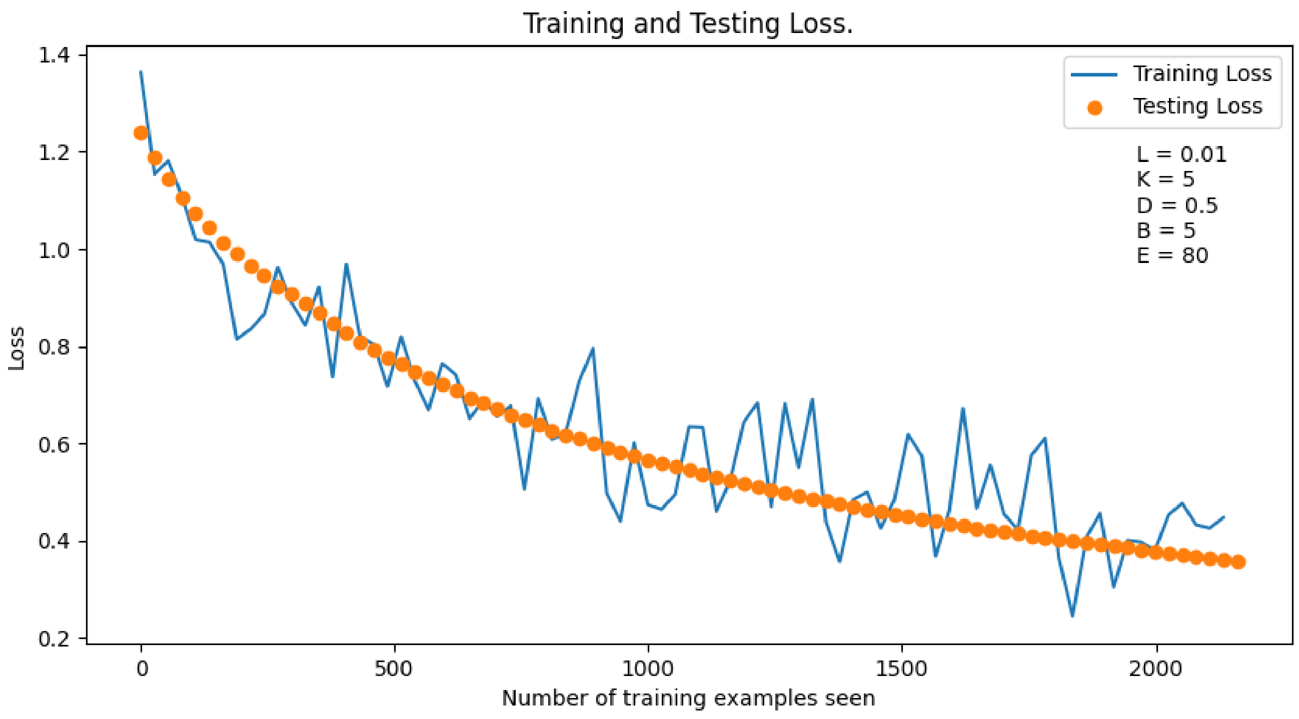
Task 3:

We used the same MNIST network that we developed for task 1. We replaced the last layer with a new layer with 3 nodes to identify the three Greek letters, alpha, beta and gamma. We trained the network and recorded the training error. We tested for this network and got a plot for the same. Then we added additional images for the same three letters using handwritten letters.
Print of model structure with new layer with 3 nodes:

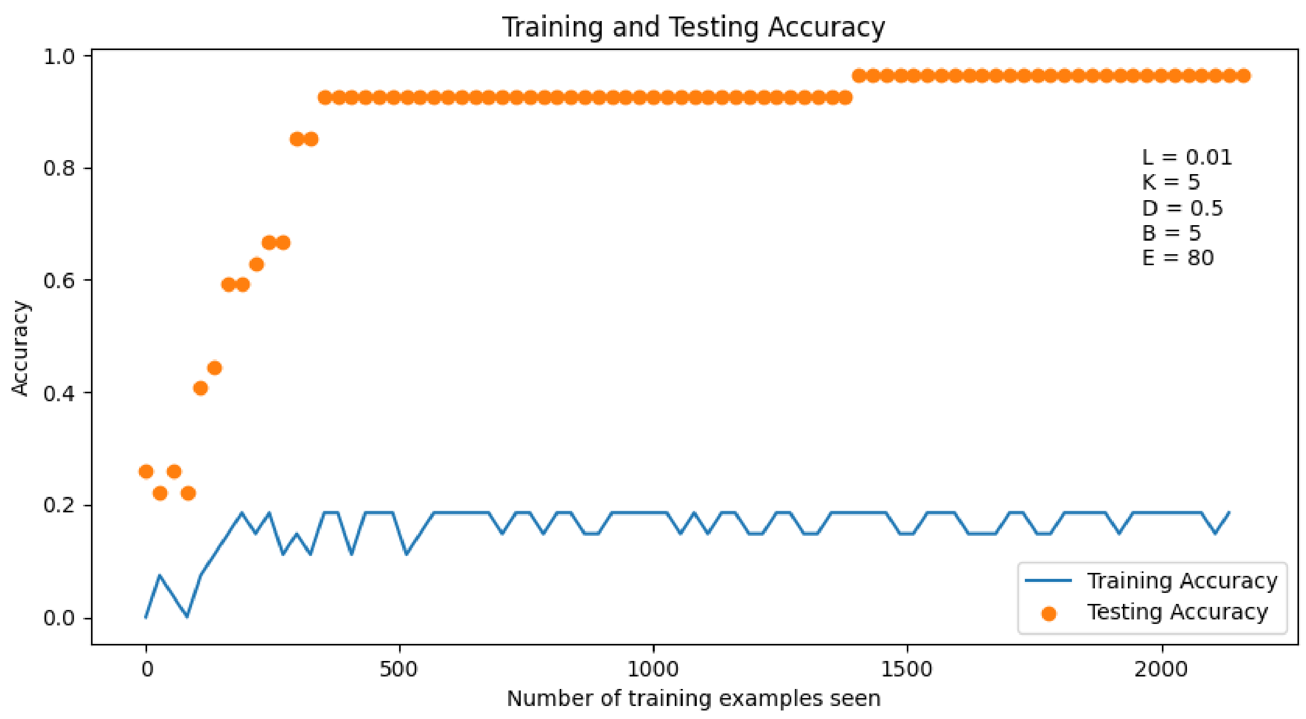


```
C:\Python310\python.exe "C:/Users/m2mah/Documents/Projects/5330/Project/Project 5/task_3_ext_2.py"
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu1): ReLU()
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout(p=0.5, inplace=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu2): ReLU()
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=50, out_features=3, bias=True)
  (log_softmax): LogSoftmax(dim=1)
)
```

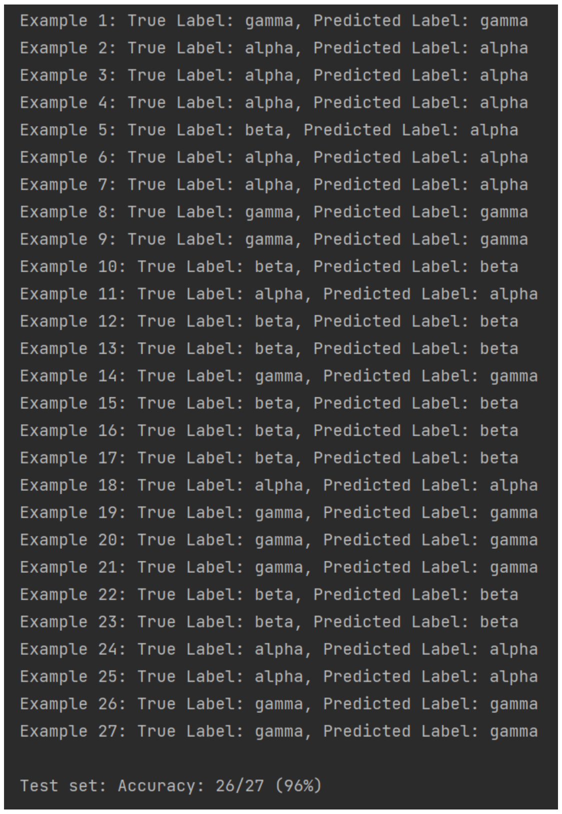
Plot of training and test loss:



Plot of test and training accuracy:

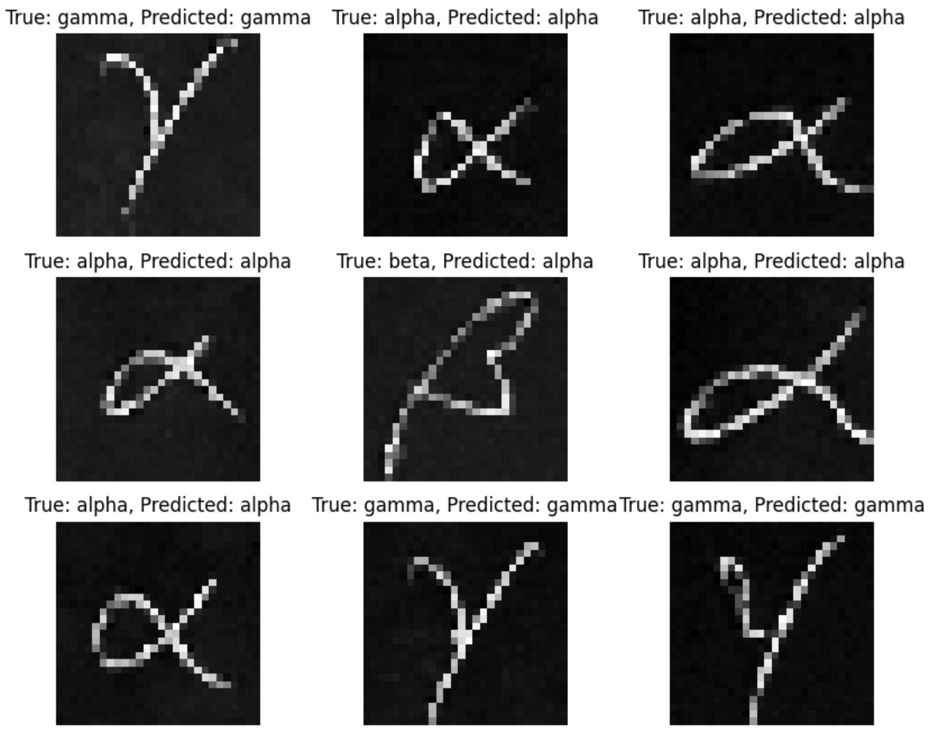


Test results and accuracy on given greek characters dataset: We observe a 26/27 (96%) accuracy of the model on the given greek dataset after training the last layer for 80 epochs.

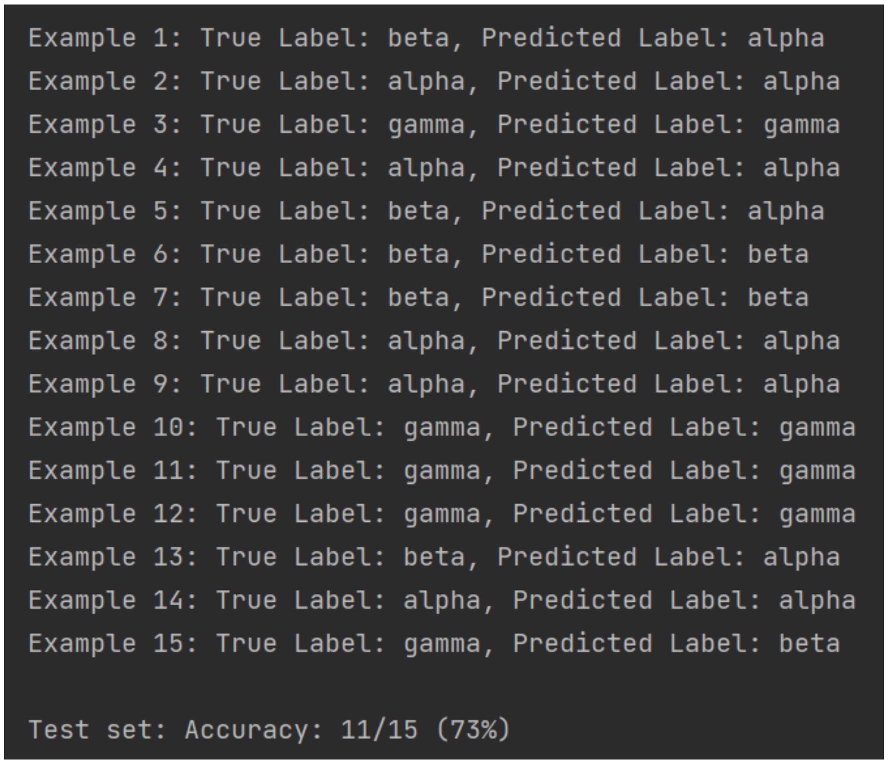


```
Example 1: True Label: gamma, Predicted Label: gamma
Example 2: True Label: alpha, Predicted Label: alpha
Example 3: True Label: alpha, Predicted Label: alpha
Example 4: True Label: alpha, Predicted Label: alpha
Example 5: True Label: beta, Predicted Label: alpha
Example 6: True Label: alpha, Predicted Label: alpha
Example 7: True Label: alpha, Predicted Label: alpha
Example 8: True Label: gamma, Predicted Label: gamma
Example 9: True Label: gamma, Predicted Label: gamma
Example 10: True Label: beta, Predicted Label: beta
Example 11: True Label: alpha, Predicted Label: alpha
Example 12: True Label: beta, Predicted Label: beta
Example 13: True Label: beta, Predicted Label: beta
Example 14: True Label: gamma, Predicted Label: gamma
Example 15: True Label: beta, Predicted Label: beta
Example 16: True Label: beta, Predicted Label: beta
Example 17: True Label: beta, Predicted Label: beta
Example 18: True Label: alpha, Predicted Label: alpha
Example 19: True Label: gamma, Predicted Label: gamma
Example 20: True Label: gamma, Predicted Label: gamma
Example 21: True Label: gamma, Predicted Label: gamma
Example 22: True Label: beta, Predicted Label: beta
Example 23: True Label: beta, Predicted Label: beta
Example 24: True Label: alpha, Predicted Label: alpha
Example 25: True Label: alpha, Predicted Label: alpha
Example 26: True Label: gamma, Predicted Label: gamma
Example 27: True Label: gamma, Predicted Label: gamma

Test set: Accuracy: 26/27 (96%)
```

Visual plot of test samples from given dataset:

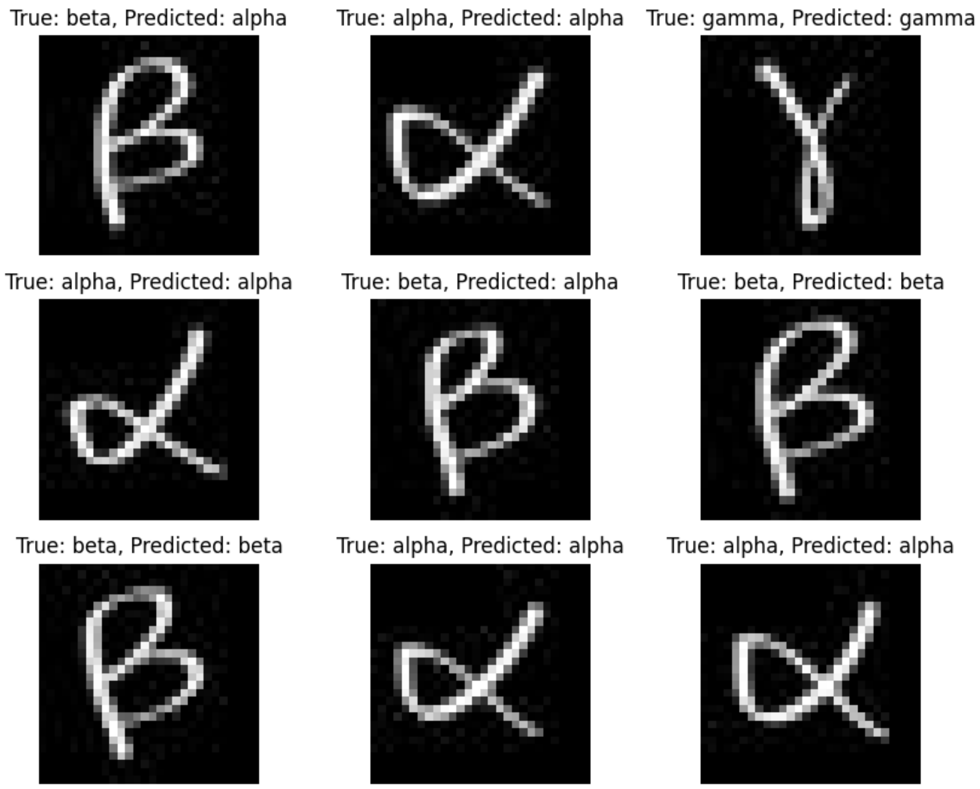

True: gamma, Predicted: gamma
True: alpha, Predicted: alpha
True: alpha, Predicted: alpha
True: alpha, Predicted: alpha
True: beta, Predicted: alpha
True: alpha, Predicted: alpha
True: alpha, Predicted: alpha
True: gamma, Predicted: gamma
True: gamma, Predicted: gamma

Test results and accuracy on new handwritten greek characters dataset: We observe 73% accuracy on the greek character dataset that we created.



```
Example 1: True Label: beta, Predicted Label: alpha
Example 2: True Label: alpha, Predicted Label: alpha
Example 3: True Label: gamma, Predicted Label: gamma
Example 4: True Label: alpha, Predicted Label: alpha
Example 5: True Label: beta, Predicted Label: alpha
Example 6: True Label: beta, Predicted Label: beta
Example 7: True Label: beta, Predicted Label: beta
Example 8: True Label: alpha, Predicted Label: alpha
Example 9: True Label: alpha, Predicted Label: alpha
Example 10: True Label: gamma, Predicted Label: gamma
Example 11: True Label: gamma, Predicted Label: gamma
Example 12: True Label: gamma, Predicted Label: gamma
Example 13: True Label: beta, Predicted Label: alpha
Example 14: True Label: alpha, Predicted Label: alpha
Example 15: True Label: gamma, Predicted Label: beta

Test set: Accuracy: 11/15 (73%)
```

Visual plot of test samples from new handwritten greek characters dataset:


True: beta, Predicted: alpha
True: alpha, Predicted: alpha
True: gamma, Predicted: gamma
True: alpha, Predicted: alpha
True: beta, Predicted: alpha
True: beta, Predicted: beta
True: beta, Predicted: beta
True: alpha, Predicted: alpha
True: alpha, Predicted: alpha

Task 4:

For this task, we loaded the Fashion MNIST dataset and trained it in a similar manner. We automated the process of evaluating different hyperparameters by creating arrays of values for each hyperparameter and dynamically create and train the model to measure it's performance. The dimensions we chose to evaluate are for dropout rate, learning rate and number of epochs of training. We chose the following sets of values to be used for evaluating the model:

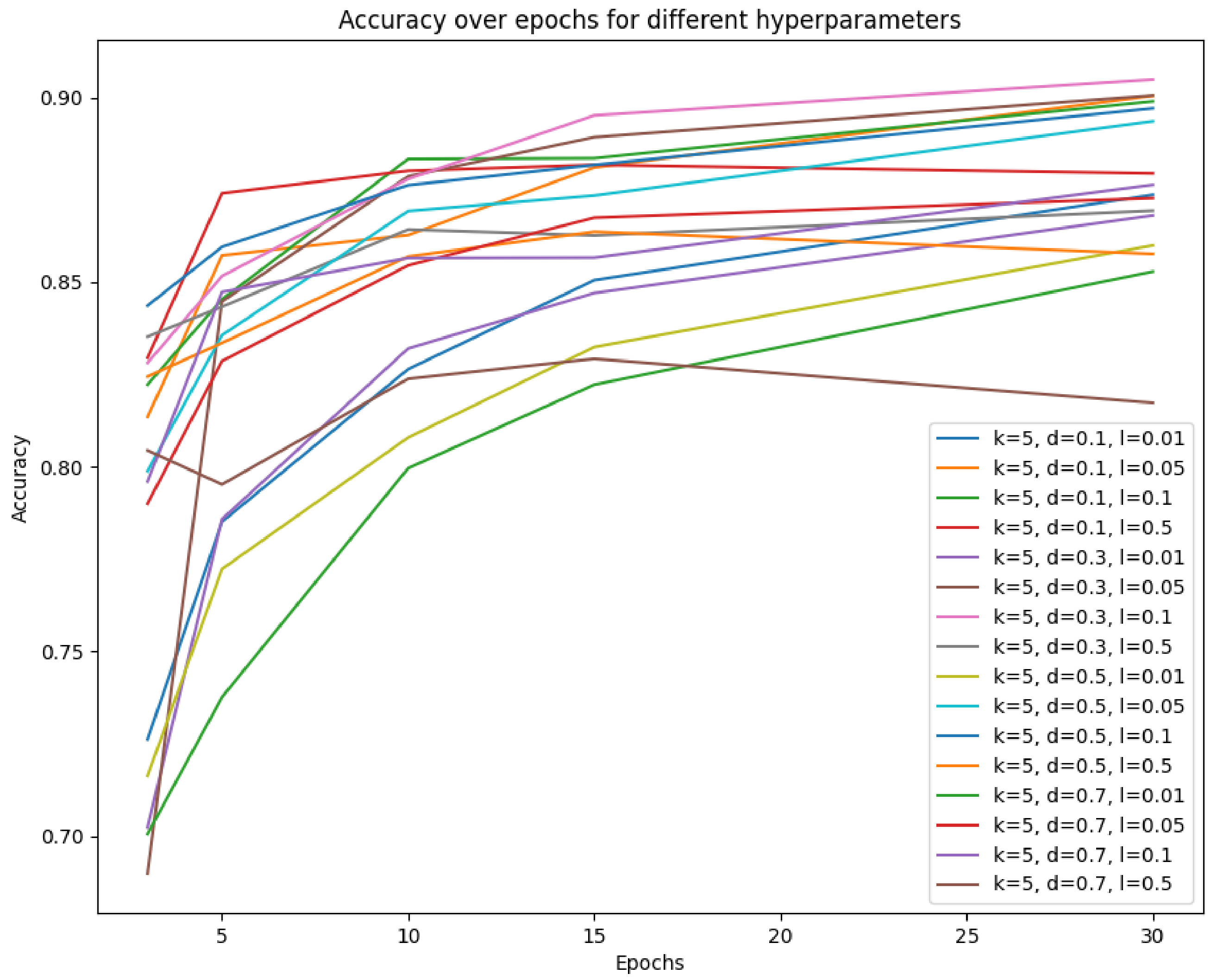
- Dropout rate: [0.1, 0.3, 0.5, 0.7]
- Learning rate: [0.01, 0.05, 0.1, 0.5]
- Epochs: [3, 5, 10, 15, 30]

Thus we would evaluate 4 * 4 * 5 variations of model hyperparameters.
Hypothesis:

1. For the number of epochs we hypothesize, "The more epochs we train, the better accuracy we can expect from the model".
2. Learning rate: A learning rate that is too low or too high will cause problems. A learning rate that is too low will take more time to learn, while learning rates too high will not allow convergence.
3. Dropout rate: A dropout rate that is too high could cause the model to not learn parameters effectively. While dropout rate that is too low could cause the model to overfit training data leading to poor test data accuracy due to lack of generalization.

We keep all other parameters the same as in the original model from Task 1.C.

Following is a plot of all combinations of hyperparameters tested:

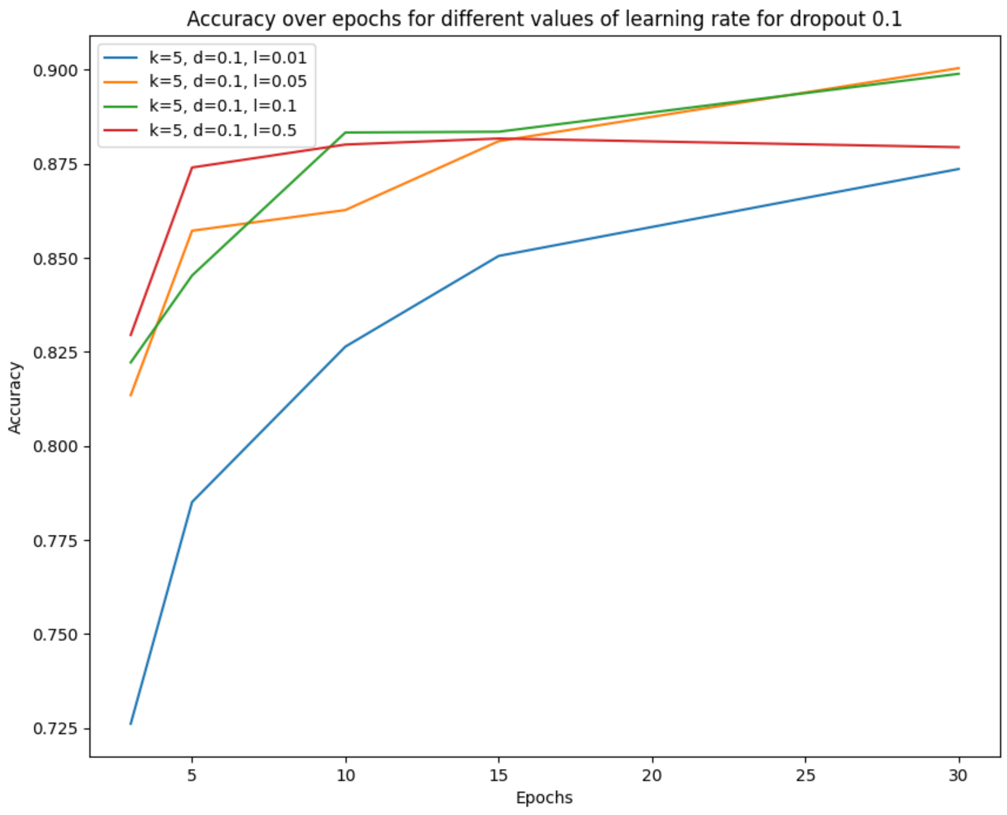


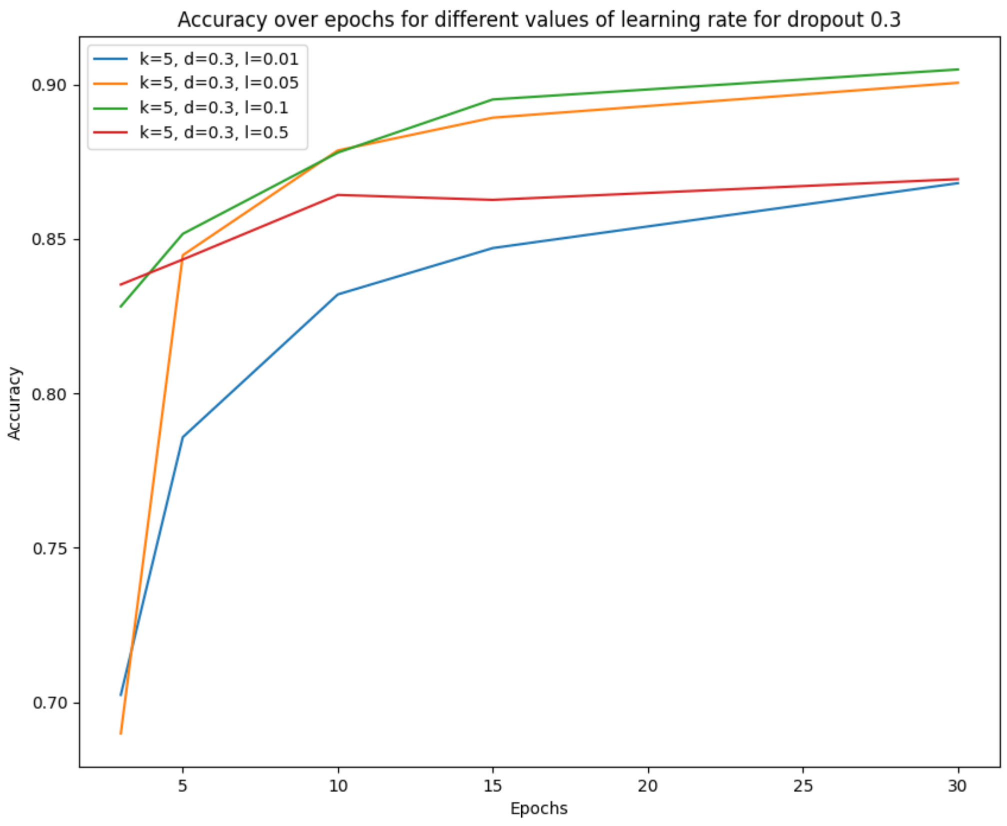
From the above plot we see that for most model configurations increasing the number of epochs of training increases the test accuracy. This is in accordance with out hypothesis about number of epochs.

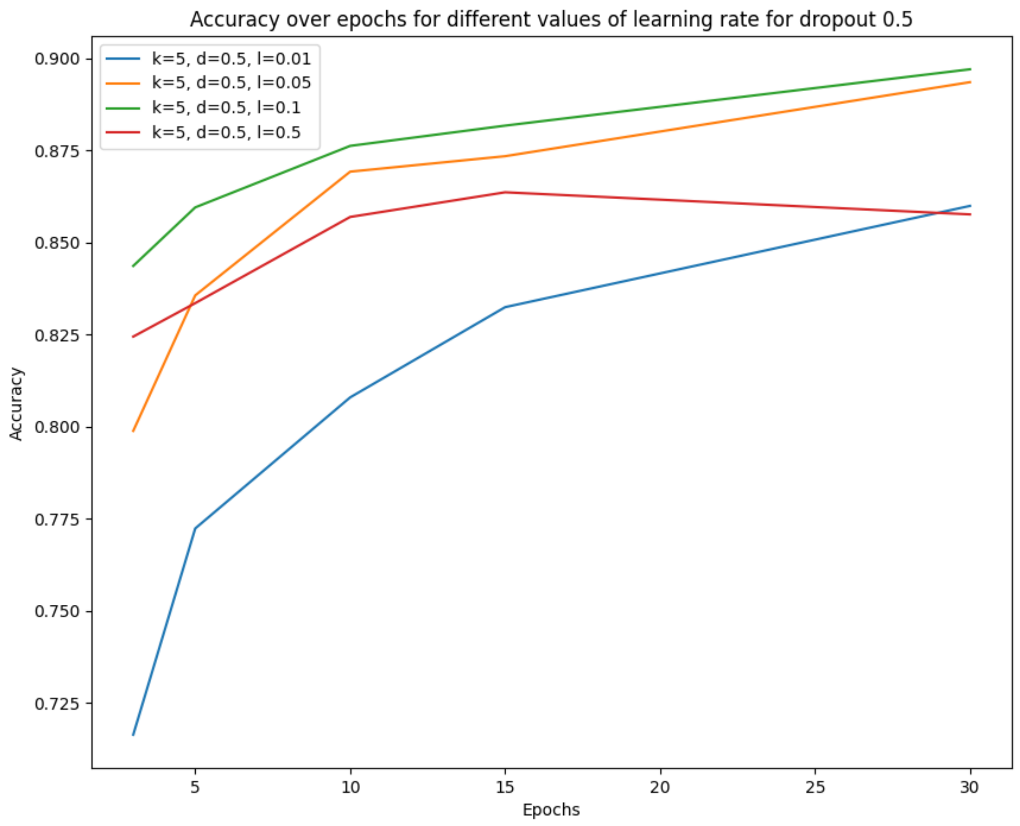
However, we observe peculiar behavior in 3 line plots where increasing the number of epochs led to a decrease in test accuracy. These configurations had a high learning rate of 0.5. Which means the high learning fails to converge and instead causes divergence after peaking in accuracy at an earlier epoch.

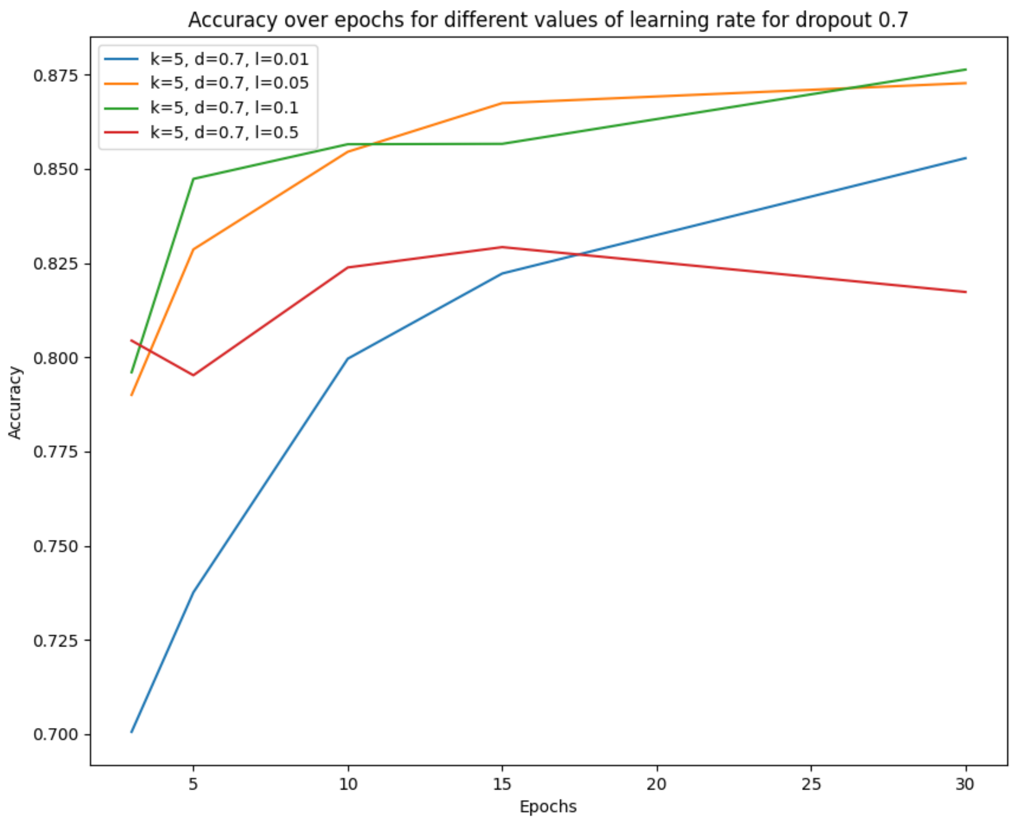
Another observation is that models with low dropout rates see an immediate jump in accuracy in earlier epochs while higher dropout rates cause the accuracy to increase gradually.

Next we plot the graphs for a constant dropout rate with a changing learning rate.

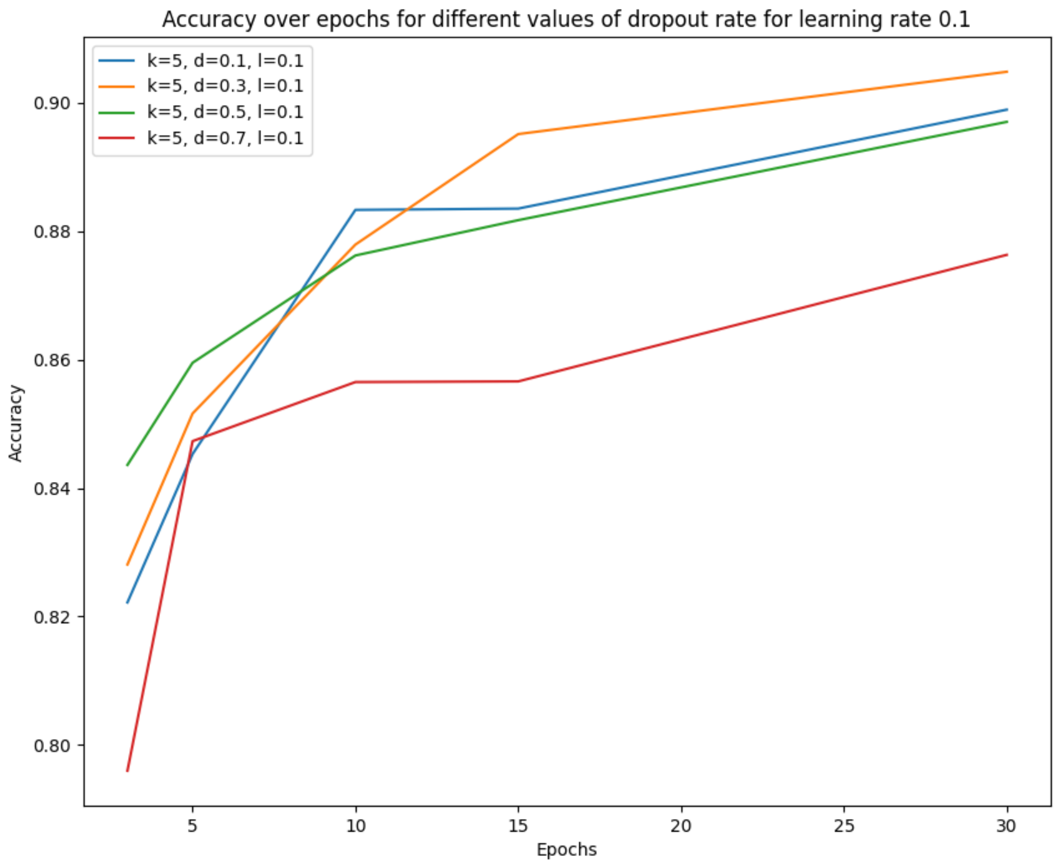





Accuracy over epochs for different values of learning rate for dropout 0.5
k=5, d=0.5, l=0.01
k=5, d=0.5, l=0.05
k=5, d=0.5, l=0.1
k=5, d=0.5, l=0.5
Accuracy
Epochs
0.900
0.875
0.850
0.825
0.800
0.775
0.750
0.725
5
10
15
20
25
30


Accuracy over epochs for different values of learning rate for dropout 0.7
k=5, d=0.7, l=0.01
k=5, d=0.7, l=0.05
k=5, d=0.7, l=0.1
k=5, d=0.7, l=0.5
Accuracy
Epochs
0.875
0.850
0.825
0.800
0.775
0.750
0.725
0.700
5
10
15
20
25
30

We observe that low learning rates (0.01) show gradual increase in accuracy but they still have very low accuracy by epoch 30. And the high learning rate (0.5) causes lower accuracy after a peak. The best performing values for learning rate seem to be 0.05 and 0.1 which lie in the middle. This is in accordance with our hypothesis about the learning rate.
We observe that 0.1 is the better value among the 2. So we plot the graph for different dropout rates across epochs while keeping constant learning rate of 0.1.



We observe that a higher dropout rate leads to lower accuracy. While low (0.1) and high (0.5) give similar results in accuracy, a very high dropout rate (0.7) performs significantly worse. Meanwhile a value in the middle 0.3 gives us the best accuracy in 30 epochs. This observation is also in accordance with our hypothesis.
Overall, we find that a model with kernel size 5, dropout rate 0.3 and learning rate 0.1 gave us the best results after training for 15 epochs and continuing the same till 30 epochs.

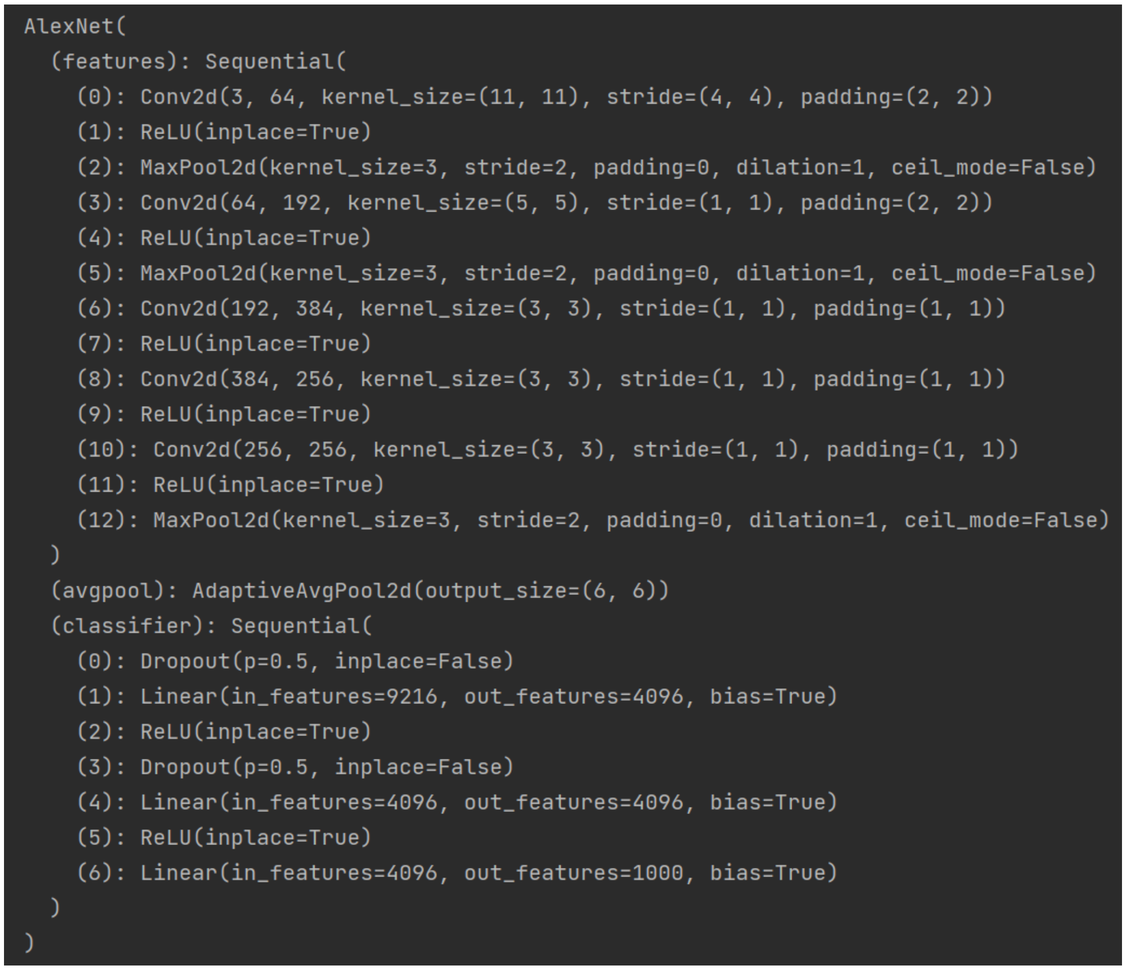
Extensions:

For extensions we decided to perform the following tasks:

Extension 1: Loading a pre trained model and evaluating its first couple of convolutional layers as in task 2.

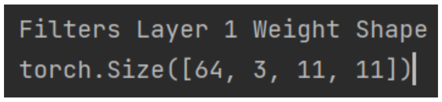
We decided to use the AlexNet model with pretrained weights from the list of available models in pytorch.

Following is the structure of the model:



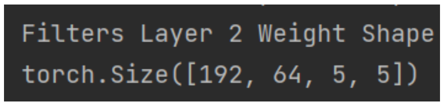
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

It has 5 convolutional layers. We take a look at the first 2 convolutional layers for our analysis. We observe that the model allows 3 color channel input and has 64 filters with kernel size 11 in the first convolutional layer.



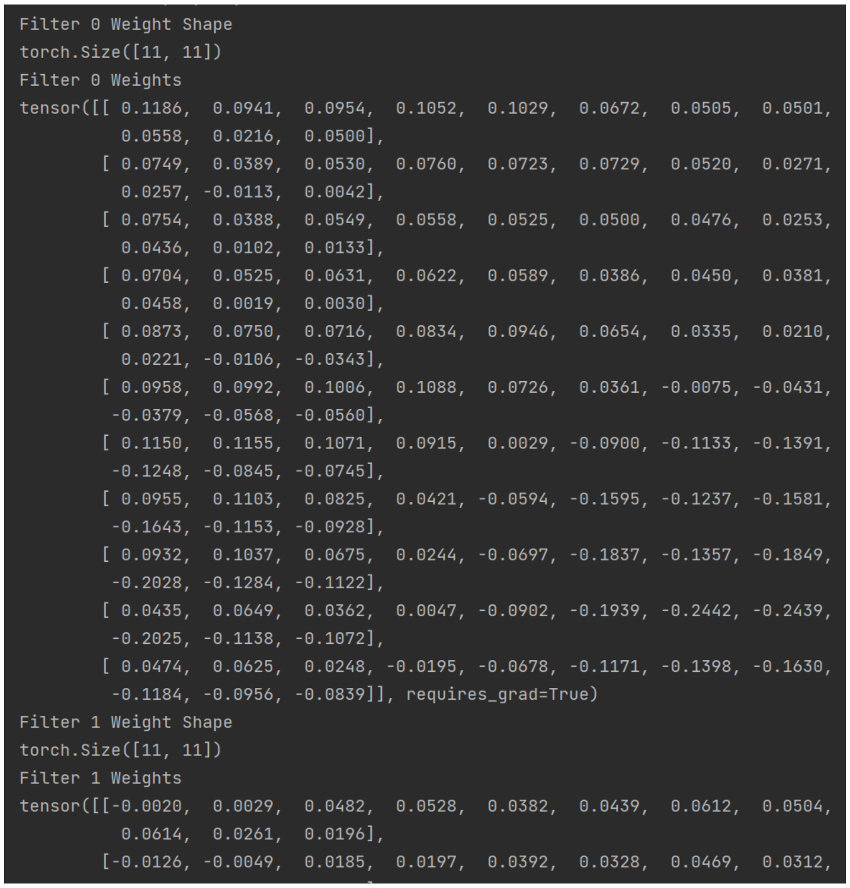
```
Filters Layer 1 Weight Shape
torch.Size([64, 3, 11, 11])
```

The second convolutional layer has 192 filters with the kernel size 5. It takes the outputs of the first convolutional layer as it's input.



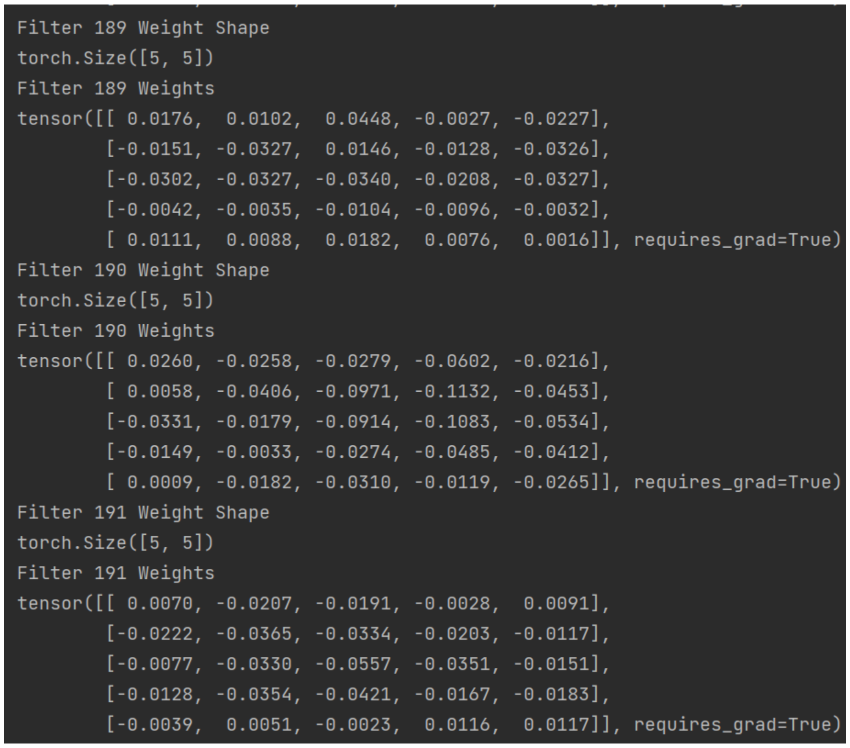
```
Filters Layer 2 Weight Shape
torch.Size([192, 64, 5, 5])
```

Sample of weights of first convolution layer printed on console and their shape:



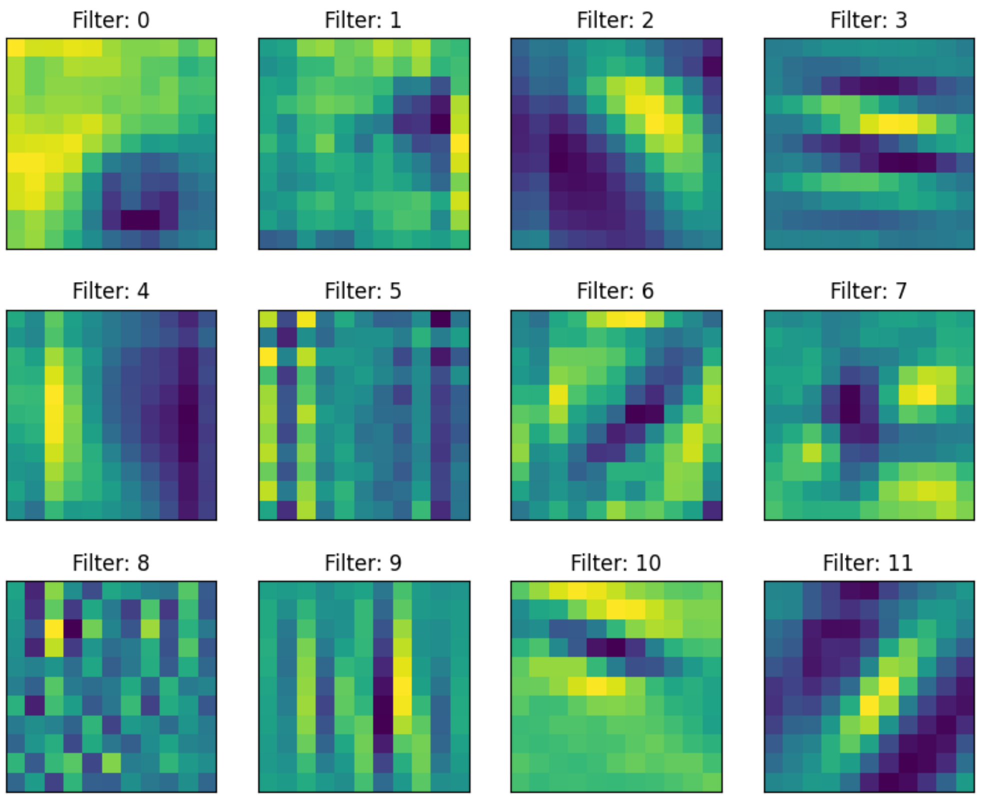
```
Filter 0 Weight Shape
torch.Size([11, 11])
Filter 0 Weights
tensor([[ 0.1186,  0.0941,  0.0954,  0.1052,  0.1029,  0.0672,  0.0505,  0.0501,
          0.0558,  0.0216,  0.0500],
        [ 0.0749,  0.0389,  0.0530,  0.0760,  0.0723,  0.0729,  0.0520,  0.0271,
          0.0257, -0.0113,  0.0042],
        [ 0.0754,  0.0388,  0.0549,  0.0558,  0.0525,  0.0500,  0.0476,  0.0253,
          0.0436,  0.0102,  0.0133],
        [ 0.0704,  0.0525,  0.0631,  0.0622,  0.0589,  0.0386,  0.0450,  0.0381,
          0.0458,  0.0019,  0.0030],
        [ 0.0873,  0.0750,  0.0716,  0.0834,  0.0946,  0.0654,  0.0335,  0.0210,
          0.0221, -0.0106, -0.0343],
        [ 0.0958,  0.0992,  0.1006,  0.1088,  0.0726,  0.0361, -0.0075, -0.0431,
         -0.0379, -0.0568, -0.0560],
        [ 0.1150,  0.1155,  0.1071,  0.0915,  0.0029, -0.0900, -0.1133, -0.1391,
         -0.1248, -0.0845, -0.0745],
        [ 0.0955,  0.1103,  0.0825,  0.0421, -0.0594, -0.1595, -0.1237, -0.1581,
         -0.1643, -0.1153, -0.0928],
        [ 0.0932,  0.1037,  0.0675,  0.0244, -0.0697, -0.1837, -0.1357, -0.1849,
         -0.2028, -0.1284, -0.1122],
        [ 0.0435,  0.0649,  0.0362,  0.0047, -0.0902, -0.1939, -0.2442, -0.2439,
         -0.2025, -0.1138, -0.1072],
        [ 0.0474,  0.0625,  0.0248, -0.0195, -0.0678, -0.1171, -0.1398, -0.1630,
         -0.1184, -0.0956, -0.0839]], requires_grad=True)
Filter 1 Weight Shape
torch.Size([11, 11])
Filter 1 Weights
tensor([[-0.0020,  0.0029,  0.0482,  0.0528,  0.0382,  0.0439,  0.0612,  0.0504,
          0.0614,  0.0261,  0.0196],
        [-0.0126, -0.0049,  0.0185,  0.0197,  0.0392,  0.0328,  0.0469,  0.0312,
```

Sample of weights of second convolution layer printed on console and their shape:

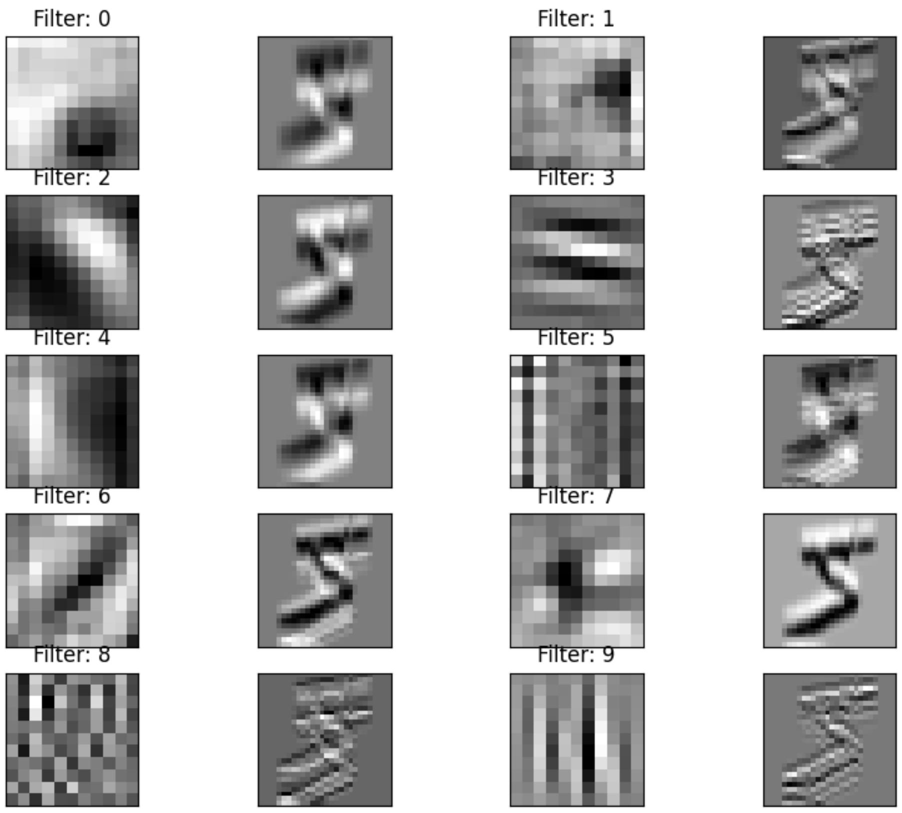


```
Filter 189 Weight Shape
torch.Size([5, 5])
Filter 189 Weights
tensor([[ 0.0176,  0.0102,  0.0448, -0.0027, -0.0227],
        [-0.0151, -0.0327,  0.0146, -0.0128, -0.0326],
        [-0.0302, -0.0327, -0.0340, -0.0208, -0.0327],
        [-0.0042, -0.0035, -0.0104, -0.0096, -0.0032],
        [ 0.0111,  0.0088,  0.0182,  0.0076,  0.0016]], requires_grad=True)
Filter 190 Weight Shape
torch.Size([5, 5])
Filter 190 Weights
tensor([[ 0.0260, -0.0258, -0.0279, -0.0602, -0.0216],
        [ 0.0058, -0.0406, -0.0971, -0.1132, -0.0453],
        [-0.0331, -0.0179, -0.0914, -0.1083, -0.0534],
        [-0.0149, -0.0033, -0.0274, -0.0485, -0.0412],
        [ 0.0009, -0.0182, -0.0310, -0.0119, -0.0265]], requires_grad=True)
Filter 191 Weight Shape
torch.Size([5, 5])
Filter 191 Weights
tensor([[ 0.0070, -0.0207, -0.0191, -0.0028,  0.0091],
        [-0.0222, -0.0365, -0.0334, -0.0203, -0.0117],
        [-0.0077, -0.0330, -0.0557, -0.0351, -0.0151],
        [-0.0128, -0.0354, -0.0421, -0.0167, -0.0183],
        [-0.0039,  0.0051, -0.0023,  0.0116,  0.0117]], requires_grad=True)
```

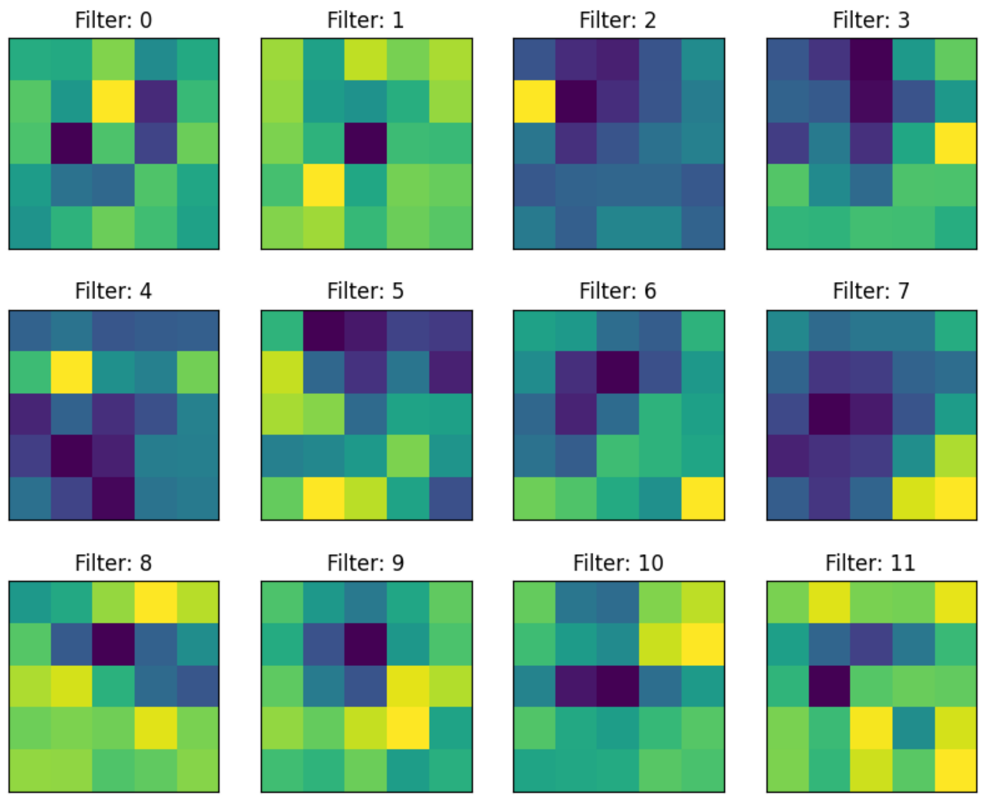
Visualization of first 12 filters of layer 1:



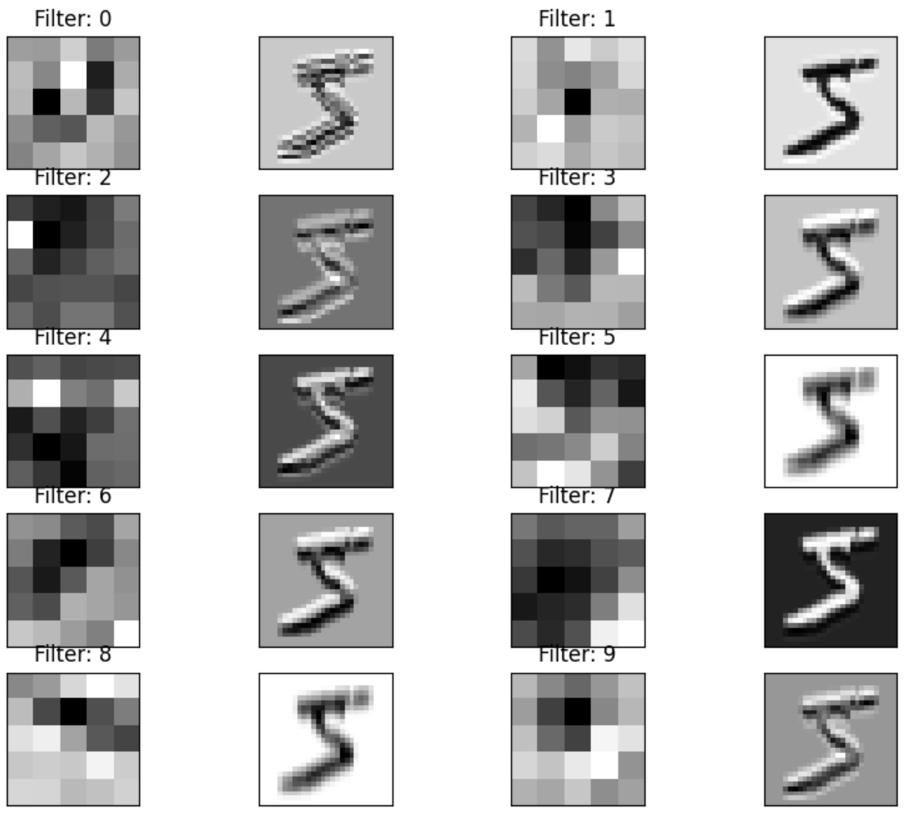
Plot of the filters and the corresponding filtered image:



Visualization of first 12 filters of layer 2:



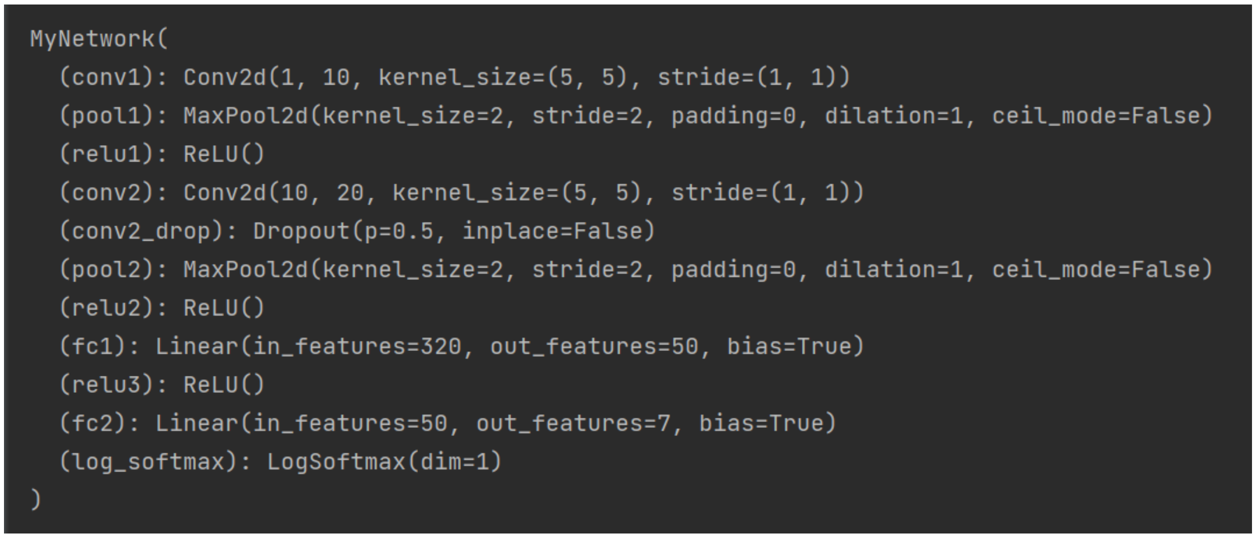
Plot of the filters and the corresponding filtered image:



Extension 2: Try more greek letters than alpha, beta and gamma

For this extension, we trained the greek letter model with additional greek alphabets theta, mu, eta and delta. We created 10 images of each alphabet for the training. We modified the output layer to accommodate for the new classes.

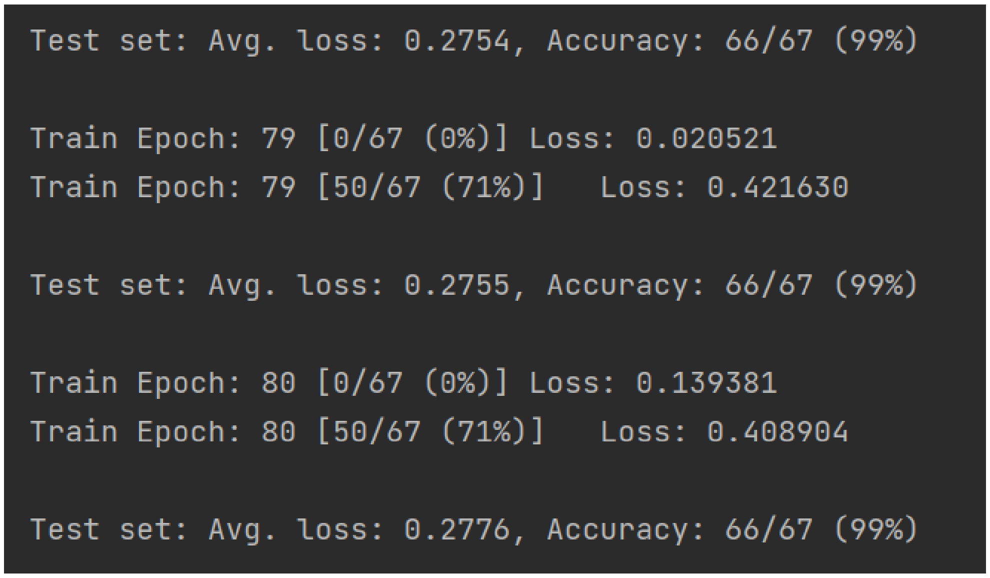
Following is the new structure of the model:



```
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu1): ReLU()
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout(p=0.5, inplace=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu2): ReLU()
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=50, out_features=7, bias=True)
  (log_softmax): LogSoftmax(dim=1)
)
```

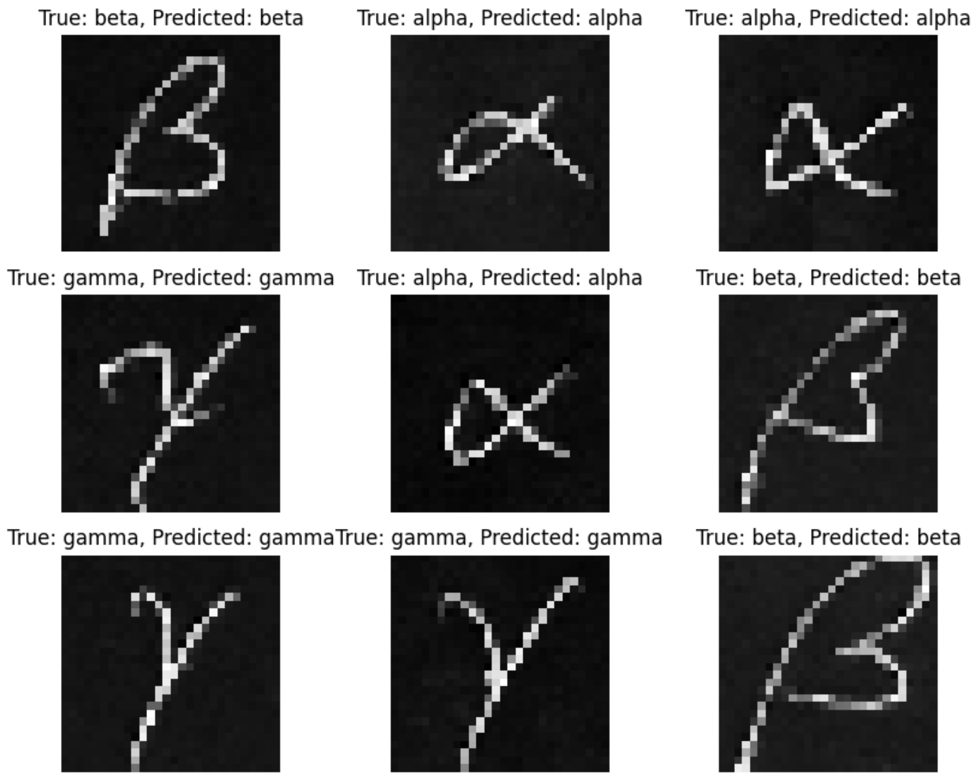
We trained the last layer by combining the greek_train and our custom dataset using a concat dataset in pytorch.

After training the model for 80 epochs we were able to achieve 99% accuracy using the combined dataset for testing:

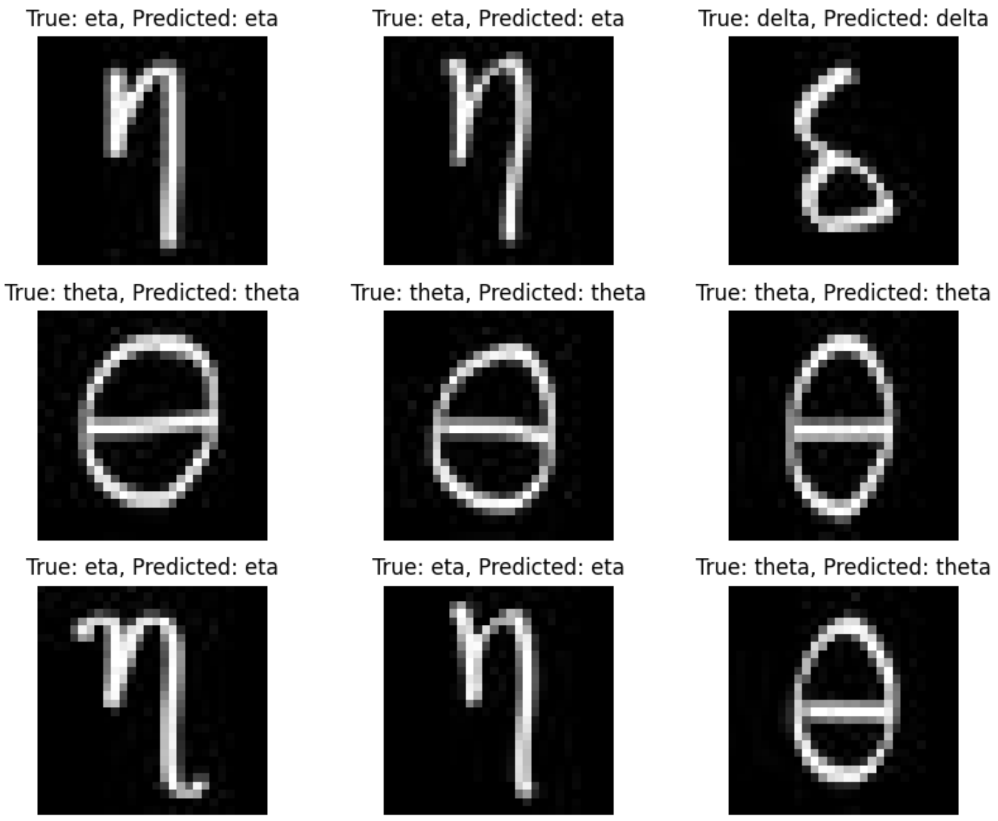


```
Test set: Avg. loss: 0.2754, Accuracy: 66/67 (99%)

Train Epoch: 79 [0/67 (0%)] Loss: 0.020521
Train Epoch: 79 [50/67 (71%)]   Loss: 0.421630

Test set: Avg. loss: 0.2755, Accuracy: 66/67 (99%)

Train Epoch: 80 [0/67 (0%)] Loss: 0.139381
Train Epoch: 80 [50/67 (71%)]   Loss: 0.408904

Test set: Avg. loss: 0.2776, Accuracy: 66/67 (99%)
```

Visual plot of test samples from given dataset:

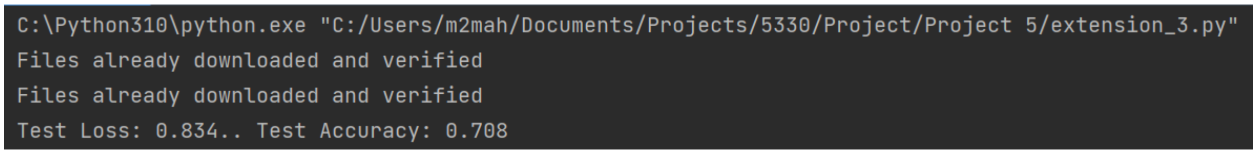

True: beta, Predicted: beta
True: alpha, Predicted: alpha
True: alpha, Predicted: alpha
True: gamma, Predicted: gamma
True: alpha, Predicted: alpha
True: beta, Predicted: beta
True: gamma, Predicted: gamma
True: gamma, Predicted: gamma
True: beta, Predicted: beta

Visual plot of test samples from new handwritten greek characters dataset:


True: eta, Predicted: eta
True: eta, Predicted: eta
True: delta, Predicted: delta
True: theta, Predicted: theta
True: theta, Predicted: theta
True: theta, Predicted: theta
True: eta, Predicted: eta
True: eta, Predicted: eta
True: theta, Predicted: theta

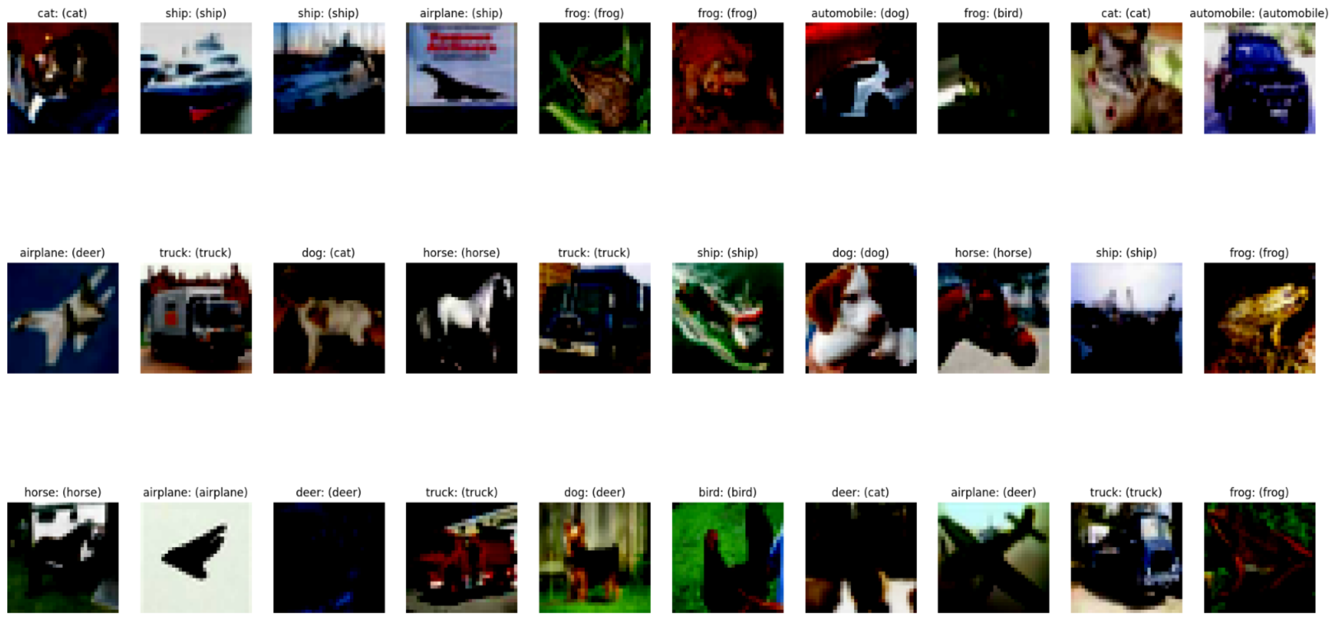
Extension 3: Explore a different computer vision task with available data

For this extension we chose to the CIFAR10 dataset from pytorch which consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
I created a model similar to task 1 but had to make necessary changes to accommodate for color channels and the new embedding size due to changes in image size.
The model was trained for 300 epochs on the data and achieved 70% accuracy on the test data.



```
C:\Python310\python.exe "C:/Users/m2mah/Documents/Projects/5330/Project/Project 5/extension_3.py"
Files already downloaded and verified
Files already downloaded and verified
Test Loss: 0.834.. Test Accuracy: 0.708
```

Visualization of some sample images from the test set:



To improve performance we could in the future build a more complex model by adding more convolutional layers, or tweaking size of filters and other hyperparameters.

Reflection:

From this project, we understood the steps taken to train a dataset and get the prediction results for a pre-existing testing set and a data set of handwritten images and compare how the network performs in both cases. We also trained the model for different data sets to get the predictions for each model and check how many epochs it takes to train different models precisely. We also compared the results for different different dimensions like changing the convolution filter size, the dropout rate, the number of epochs and the learning rate. We could analyze how these dimensions affect the prediction results and errors.

Acknowledgements:

Firstly we would like to thank Prof. Bruce Maxwell for introducing this topic and always being available to clarify any doubts we encountered. Next we would like to thank the TAs, for helping us out understand the problems and helping us with the errors we kept getting. Finally we would like to mention a few sites that helped us understand few OpenCV and C++ functions:

- Defining a Neural Network in PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation
- Building Models with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation
- Training with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation
- Learn the Basics — PyTorch Tutorials 2.0.0+cu117 documentation
- Deep Learning With PyTorch - Full Course - YouTube
- MNIST Handwritten Digit Recognition in PyTorch – Nextjournal