CSS5330 Pattern Recognition and Computer Vision

Project 5: Recognition using Deep Networks

Francis Jacob Kalliath

1. A short description of the overall project

The Recognition using deep networks project helps us understand the various stages of creating a deep learning model. Which includes data collection, model selection, fine-tuning, using pre-trained models, training, testing, and plotting the graphs to understand the performance. The project has hands-on implementation of the different layers of the convolution layer, pooling layer, fully connected layer, etc. The last experiment helps us understand how we can obtain a better model by changing the hyper parameters while training.

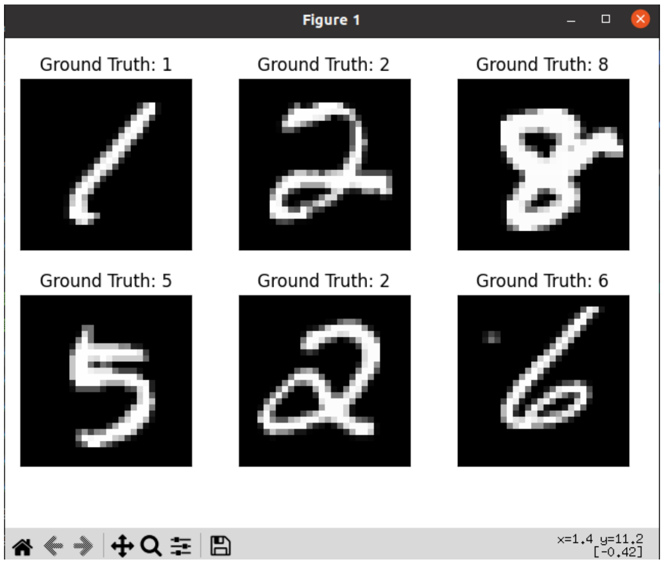
2.

TASKS:

Task 1

Build and train a network to recognize digits

A)Get the MNIST digit data set

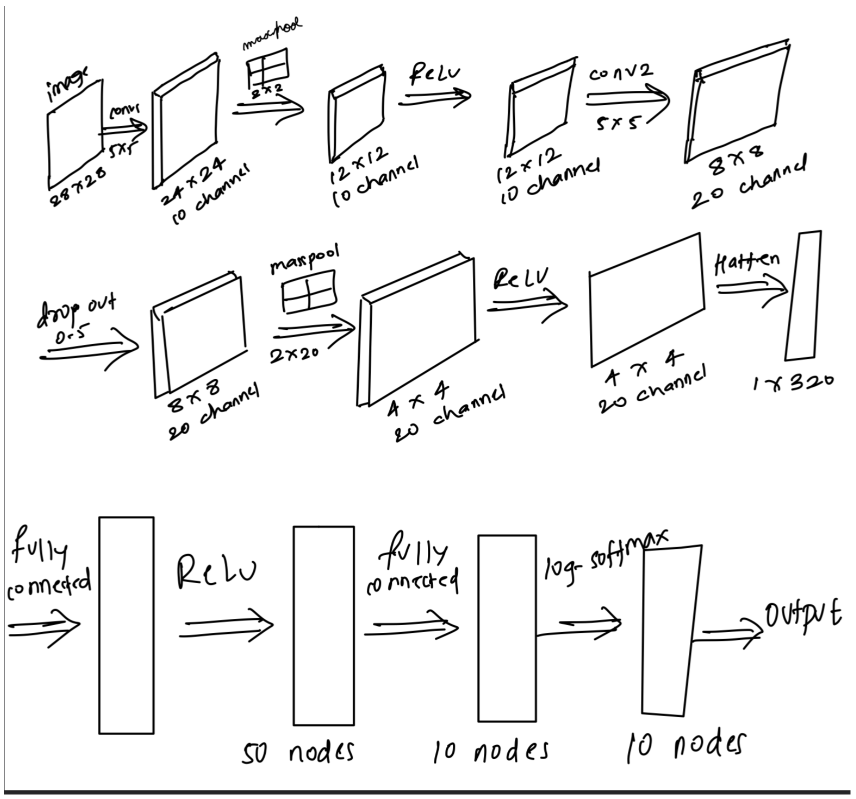


B)Make your network code repeatable

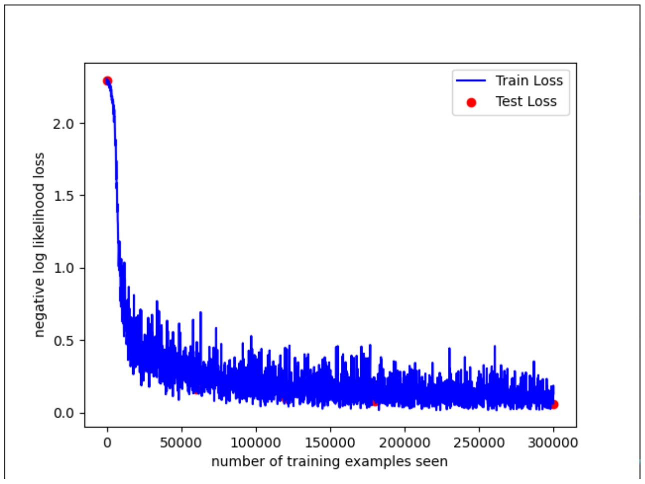
Implemented

C)Build a network model

Diagram



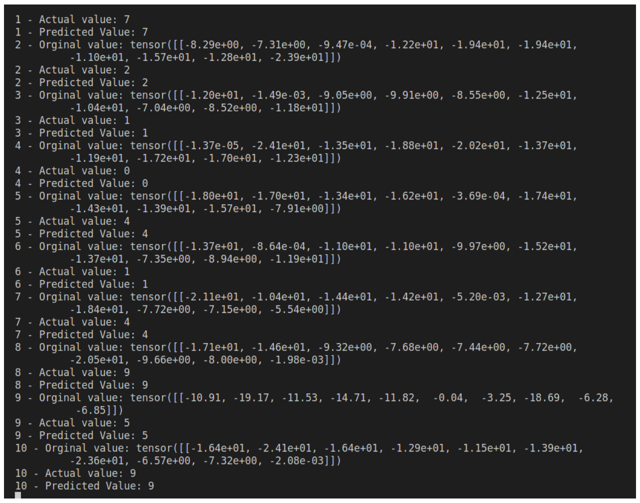
D)Train the model



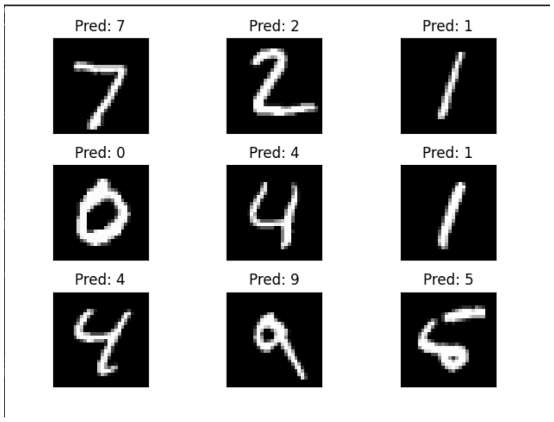
E)Save the network to a file

The network has been saved to the file

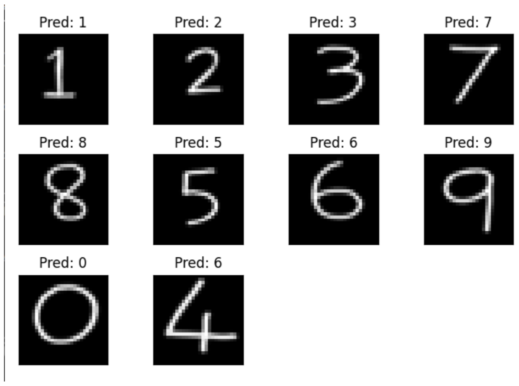
F)Read the network and run it on the test set



```
1 - Actual value: 7
1 - Predicted Value: 7
2 - Orginal value: tensor([[-8.29e+00, -7.31e+00, -9.47e-04, -1.22e+01, -1.94e+01, -1.94e+01,
         -1.10e+01, -1.57e+01, -1.28e+01, -2.39e+01]])
2 - Actual value: 2
2 - Predicted Value: 2
3 - Orginal value: tensor([[-1.20e+01, -1.49e-03, -9.05e+00, -9.91e+00, -8.55e+00, -1.25e+01,
         -1.04e+01, -7.04e+00, -8.52e+00, -1.18e+01]])
3 - Actual value: 1
3 - Predicted Value: 1
4 - Orginal value: tensor([[-1.37e-05, -2.41e+01, -1.35e+01, -1.88e+01, -2.02e+01, -1.37e+01,
         -1.19e+01, -1.72e+01, -1.70e+01, -1.23e+01]])
4 - Actual value: 0
4 - Predicted Value: 0
5 - Orginal value: tensor([[-1.80e+01, -1.70e+01, -1.34e+01, -1.62e+01, -3.69e-04, -1.74e+01,
         -1.43e+01, -1.39e+01, -1.57e+01, -7.91e+00]])
5 - Actual value: 4
5 - Predicted Value: 4
6 - Orginal value: tensor([[-1.37e+01, -8.64e-04, -1.10e+01, -1.10e+01, -9.97e+00, -1.52e+01,
         -1.37e+01, -7.35e+00, -8.94e+00, -1.19e+01]])
6 - Actual value: 1
6 - Predicted Value: 1
7 - Orginal value: tensor([[-2.11e+01, -1.04e+01, -1.44e+01, -1.42e+01, -5.20e-03, -1.27e+01,
         -1.84e+01, -7.72e+00, -7.15e+00, -5.54e+00]])
7 - Actual value: 4
7 - Predicted Value: 4
8 - Orginal value: tensor([[-1.71e+01, -1.46e+01, -9.32e+00, -7.68e+00, -7.44e+00, -7.72e+00,
         -2.05e+01, -9.66e+00, -8.00e+00, -1.98e-03]])
8 - Actual value: 9
8 - Predicted Value: 9
9 - Orginal value: tensor([[-10.91, -19.17, -11.53, -14.71, -11.82,  -0.04,  -3.25, -18.69,  -6.28,
          -6.85]])
9 - Actual value: 5
9 - Predicted Value: 5
10 - Orginal value: tensor([[-1.64e+01, -2.41e+01, -1.64e+01, -1.29e+01, -1.15e+01, -1.39e+01,
         -2.36e+01, -6.57e+00, -7.32e+00, -2.08e-03]])
10 - Actual value: 9
10 - Predicted Value: 9
```

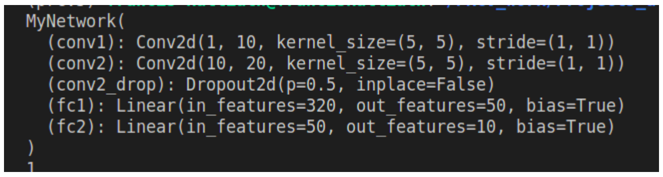


G)Test the network on new inputs



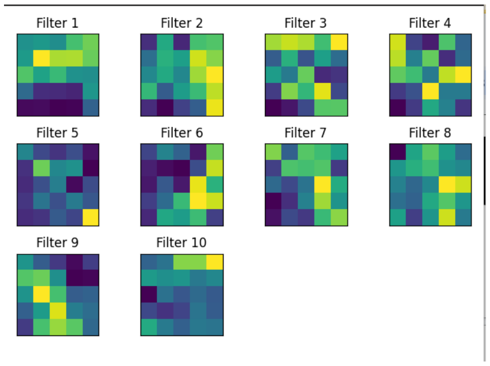
Task 2

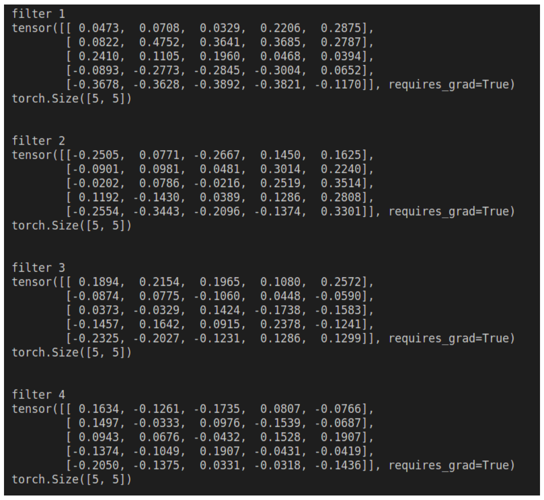
Examine your network



```
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

A)



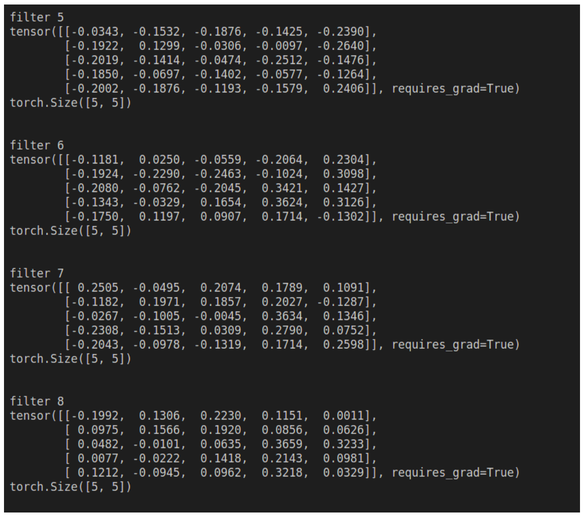


```
filter 1
tensor([[ 0.0473,  0.0708,  0.0329,  0.2206,  0.2875],
        [ 0.0822,  0.4752,  0.3641,  0.3685,  0.2787],
        [ 0.2410,  0.1105,  0.1960,  0.0468,  0.0394],
        [-0.0893, -0.2773, -0.2845, -0.3004,  0.0652],
        [-0.3678, -0.3628, -0.3892, -0.3821, -0.1170]], requires_grad=True)
torch.Size([5, 5])


filter 2
tensor([[-0.2505,  0.0771, -0.2667,  0.1450,  0.1625],
        [-0.0901,  0.0981,  0.0481,  0.3014,  0.2240],
        [-0.0202,  0.0786, -0.0216,  0.2519,  0.3514],
        [ 0.1192, -0.1430,  0.0389,  0.1286,  0.2808],
        [-0.2554, -0.3443, -0.2096, -0.1374,  0.3301]], requires_grad=True)
torch.Size([5, 5])


filter 3
tensor([[ 0.1894,  0.2154,  0.1965,  0.1080,  0.2572],
        [-0.0874,  0.0775, -0.1060,  0.0448, -0.0590],
        [ 0.0373, -0.0329,  0.1424, -0.1738, -0.1583],
        [-0.1457,  0.1642,  0.0915,  0.2378, -0.1241],
        [-0.2325, -0.2027, -0.1231,  0.1286,  0.1299]], requires_grad=True)
torch.Size([5, 5])


filter 4
tensor([[ 0.1634, -0.1261, -0.1735,  0.0807, -0.0766],
        [ 0.1497, -0.0333,  0.0976, -0.1539, -0.0687],
        [ 0.0943,  0.0676, -0.0432,  0.1528,  0.1907],
        [-0.1374, -0.1049,  0.1907, -0.0431, -0.0419],
        [-0.2050, -0.1375,  0.0331, -0.0318, -0.1436]], requires_grad=True)
torch.Size([5, 5])
```

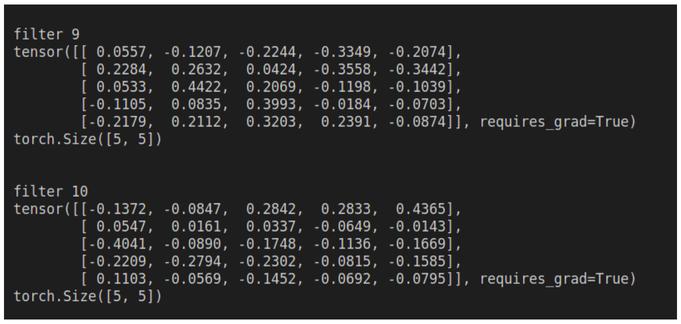


```
filter 5
tensor([[-0.0343, -0.1532, -0.1876, -0.1425, -0.2390],
        [-0.1922,  0.1299, -0.0306, -0.0097, -0.2640],
        [-0.2019, -0.1414, -0.0474, -0.2512, -0.1476],
        [-0.1850, -0.0697, -0.1402, -0.0577, -0.1264],
        [-0.2002, -0.1876, -0.1193, -0.1579,  0.2406]], requires_grad=True)
torch.Size([5, 5])


filter 6
tensor([[-0.1181,  0.0250, -0.0559, -0.2064,  0.2304],
        [-0.1924, -0.2290, -0.2463, -0.1024,  0.3098],
        [-0.2080, -0.0762, -0.2045,  0.3421,  0.1427],
        [-0.1343, -0.0329,  0.1654,  0.3624,  0.3126],
        [-0.1750,  0.1197,  0.0907,  0.1714, -0.1302]], requires_grad=True)
torch.Size([5, 5])


filter 7
tensor([[ 0.2505, -0.0495,  0.2074,  0.1789,  0.1091],
        [-0.1182,  0.1971,  0.1857,  0.2027, -0.1287],
        [-0.0267, -0.1005, -0.0045,  0.3634,  0.1346],
        [-0.2308, -0.1513,  0.0309,  0.2790,  0.0752],
        [-0.2043, -0.0978, -0.1319,  0.1714,  0.2598]], requires_grad=True)
torch.Size([5, 5])


filter 8
tensor([[-0.1992,  0.1306,  0.2230,  0.1151,  0.0011],
        [ 0.0975,  0.1566,  0.1920,  0.0856,  0.0626],
        [ 0.0482, -0.0101,  0.0635,  0.3659,  0.3233],
        [ 0.0077, -0.0222,  0.1418,  0.2143,  0.0981],
        [ 0.1212, -0.0945,  0.0962,  0.3218,  0.0329]], requires_grad=True)
torch.Size([5, 5])
```



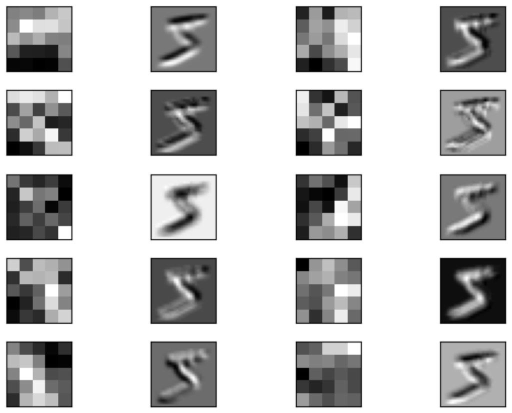
```
filter 9
tensor([[ 0.0557, -0.1207, -0.2244, -0.3349, -0.2074],
        [ 0.2284,  0.2632,  0.0424, -0.3558, -0.3442],
        [ 0.0533,  0.4422,  0.2069, -0.1198, -0.1039],
        [-0.1105,  0.0835,  0.3993, -0.0184, -0.0703],
        [-0.2179,  0.2112,  0.3203,  0.2391, -0.0874]], requires_grad=True)
torch.Size([5, 5])


filter 10
tensor([[-0.1372, -0.0847,  0.2842,  0.2833,  0.4365],
        [ 0.0547,  0.0161,  0.0337, -0.0649, -0.0143],
        [-0.4041, -0.0890, -0.1748, -0.1136, -0.1669],
        [-0.2209, -0.2794, -0.2302, -0.0815, -0.1585],
        [ 0.1103, -0.0569, -0.1452, -0.0692, -0.0795]], requires_grad=True)
torch.Size([5, 5])
```

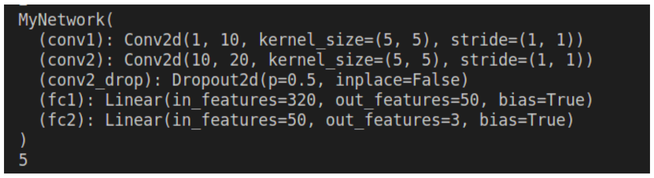
B)

The output from the first 10 filters is as follows:

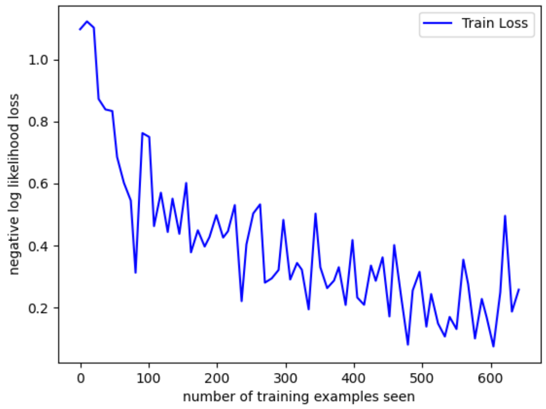
The results that are obtained are correct as the filter is accurately able to detect the edges. For instance we can observe that if there is a bright color on the top and dark color on the button side then the edges along the x axis of the object are being detected accurately. Similarly if the bright side is on the left and the dark side is on the right then we can see that the edges on the vertical axis are obtained accurately.

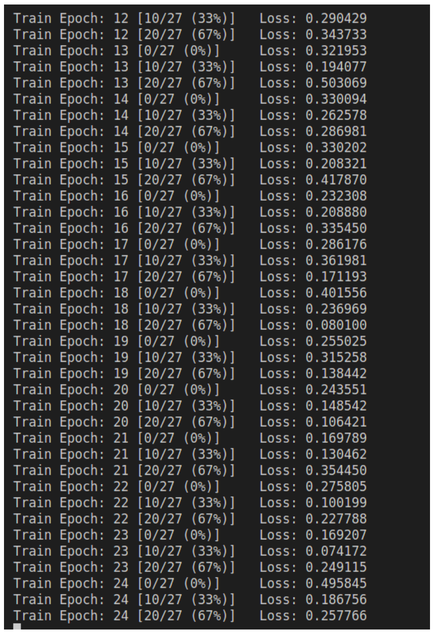


Task 3
Transfer Learning on Greek Letters



```
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=3, bias=True)
)
5
```

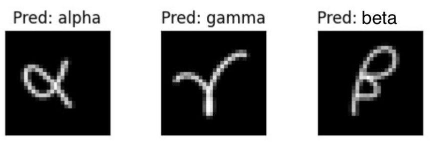




```
Train Epoch: 12 [10/27 (33%)]   Loss: 0.290429
Train Epoch: 12 [20/27 (67%)]   Loss: 0.343733
Train Epoch: 13 [0/27 (0%)]     Loss: 0.321953
Train Epoch: 13 [10/27 (33%)]   Loss: 0.194077
Train Epoch: 13 [20/27 (67%)]   Loss: 0.503069
Train Epoch: 14 [0/27 (0%)]     Loss: 0.330094
Train Epoch: 14 [10/27 (33%)]   Loss: 0.262578
Train Epoch: 14 [20/27 (67%)]   Loss: 0.286981
Train Epoch: 15 [0/27 (0%)]     Loss: 0.330202
Train Epoch: 15 [10/27 (33%)]   Loss: 0.208321
Train Epoch: 15 [20/27 (67%)]   Loss: 0.417870
Train Epoch: 16 [0/27 (0%)]     Loss: 0.232308
Train Epoch: 16 [10/27 (33%)]   Loss: 0.208880
Train Epoch: 16 [20/27 (67%)]   Loss: 0.335450
Train Epoch: 17 [0/27 (0%)]     Loss: 0.286176
Train Epoch: 17 [10/27 (33%)]   Loss: 0.361981
Train Epoch: 17 [20/27 (67%)]   Loss: 0.171193
Train Epoch: 18 [0/27 (0%)]     Loss: 0.401556
Train Epoch: 18 [10/27 (33%)]   Loss: 0.236969
Train Epoch: 18 [20/27 (67%)]   Loss: 0.080100
Train Epoch: 19 [0/27 (0%)]     Loss: 0.255025
Train Epoch: 19 [10/27 (33%)]   Loss: 0.315258
Train Epoch: 19 [20/27 (67%)]   Loss: 0.138442
Train Epoch: 20 [0/27 (0%)]     Loss: 0.243551
Train Epoch: 20 [10/27 (33%)]   Loss: 0.148542
Train Epoch: 20 [20/27 (67%)]   Loss: 0.106421
Train Epoch: 21 [0/27 (0%)]     Loss: 0.169789
Train Epoch: 21 [10/27 (33%)]   Loss: 0.130462
Train Epoch: 21 [20/27 (67%)]   Loss: 0.354450
Train Epoch: 22 [0/27 (0%)]     Loss: 0.275805
Train Epoch: 22 [10/27 (33%)]   Loss: 0.100199
Train Epoch: 22 [20/27 (67%)]   Loss: 0.227788
Train Epoch: 23 [0/27 (0%)]     Loss: 0.169207
Train Epoch: 23 [10/27 (33%)]   Loss: 0.074172
Train Epoch: 23 [20/27 (67%)]   Loss: 0.249115
Train Epoch: 24 [0/27 (0%)]     Loss: 0.495845
Train Epoch: 24 [10/27 (33%)]   Loss: 0.186756
Train Epoch: 24 [20/27 (67%)]   Loss: 0.257766
```

It takes 25 epochs to create a network to identify the Greek letters.

The predictions for a custom written greek alphabets are below:



Task 4

Design your own experiment

Change in Number of Epochs:

Increasing the number of epochs can help the model learn more complex patterns and improve its accuracy, up to a certain point. However, if the number of epochs is too high, the model can overfit to the training data, which can lead to a decrease in accuracy on new, unseen data. It is important to monitor the validation accuracy during training and choose the optimal number of epochs based on the trade-off between training accuracy and validation accuracy.

Changing Batch Size:

Increasing the batch size can speed up training and allow the model to take larger steps in the direction of the gradient, which can help it converge faster and potentially improve accuracy. However, larger batch sizes require more memory to store the intermediate activations, and they can lead to more noisy gradient estimates and slower convergence, especially for smaller datasets. On the other hand, using a smaller batch size can lead to more frequent updates of the model weights, which can result in more accurate gradient estimates and better convergence. However, smaller batch sizes can also increase the amount of time needed for training, as the model weights need to be updated more frequently.

Changing number of convolution layers
Increasing the number of convolution layers can help the model learn more complex features and improve its accuracy, up to a certain point. However, adding too many layers can lead to overfitting, especially if the model has a small dataset or if the layers are too deep relative to the input size. In general, it is recommended to start with a small number of layers and gradually increase them until the accuracy plateaus.

Changing number of filters in each convolution layer

The number of filters in each convolution layer determines the number of features that the model can learn at each layer. Increasing the number of filters can improve the model's accuracy, but it also increases the number of parameters and the computational cost of the model. Therefore, it is important to balance the number of filters with the complexity of the task and the available computational resources.

Changing Dropout Rate

The dropout rate is a regularization technique that randomly drops out some units in the network during training, which can prevent overfitting and improve the model's accuracy on new data. Increasing the dropout rate can improve the model's generalization ability, but it can also decrease the accuracy on the training data, especially if the model is not overfitting to begin with. Therefore, it is important to monitor the training and validation accuracy during training and choose an optimal dropout rate that balances the trade-off between overfitting and underfitting.

All graphs attached in a folder.

Extension 1

Replace the build model with a pre-trained model.
The Models that I have implemented are

1) VGG11
2) Alexnet
3) Dense121
4) Mobilenet_v2

Extension 2

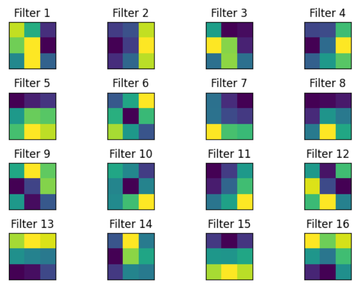
Compared the first two convolution layers of all the 4 pre-trained models.
Based on the comparison of the first two convolutional layers of the VGG11, AlexNet, DenseNet121, and MobileNetV2 models, we can see that each model has a unique architecture with different numbers of filters, kernel sizes, and pooling sizes. The VGG11 and AlexNet models have a larger number of filters and kernel sizes in their first layers, whereas the DenseNet121 and MobileNetV2 models have smaller kernel sizes and fewer filters in their first layers. This can be attributed to the fact that the former models were developed before the latter models and were designed to capture more complex features in the images.

Extension 3

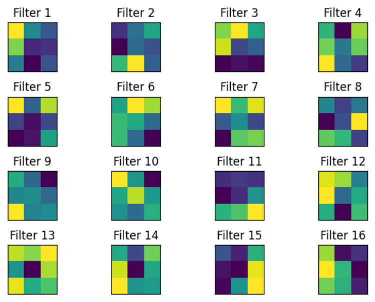
Comparing the first and forth convolution layers of the VGG11 pre-trained model. From the plots we can notice that the difference between first and fourth convolutional layers is the number of filters. The fourth layer has more filters than the first layer, which can capture more complex features. As we move deeper into the network, the number of filters typically increases to capture more abstract features.

All the results are plotted below:

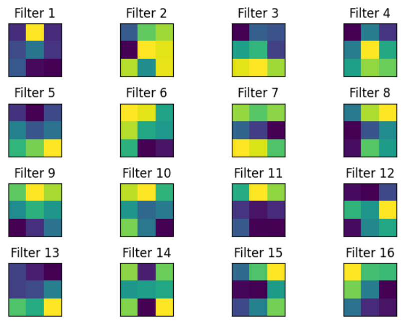
VGG11 1st convolution layer



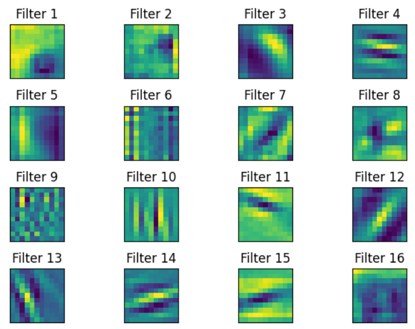
VGG11 2st convolution layer



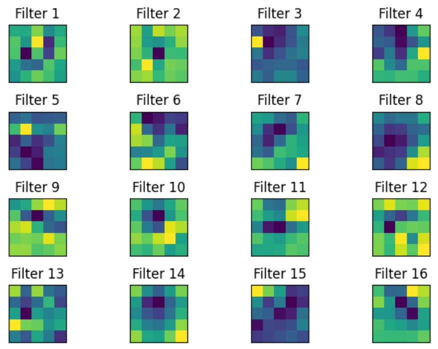
VGG11 4th convolution layer



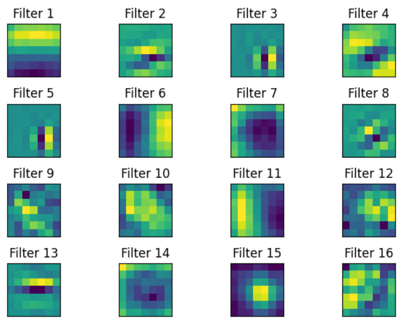
Alexnet 1st convolution layer



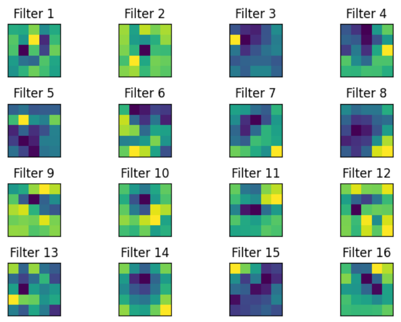
Alexnet 2nd convolution layer



Dense121 1st convolution layer



Mobilenet_v2 1st convolution layer



3. Reflection of the learnings

In this project I got hands-on exposure to object recognition using deep learning. Developing the network from scratch to using predefined models gave me a scratch step by step process of how the deep learning models work and how their accuracies can be improved by changing the hyper parameters :
Some of the most important learning from this project are:

1) In this project we developed a model from scratch, used the model and fine tuned it. Used various datasets to train the model and test it.
2) Run a loop to compare the performance of various hyper parameters and get a generalized conclusion from the test results.
3) I have also used the pretrained models from VGG11, Alexnet, Dense121, Mobilenet_v2 and compared the various convolution layers

4. Acknowledgement of the material

- HackerRank
- OpenCV Documentation / Tutorials
- Stackoverflow
- GeeksforGeeks
- W3 School
- Git
- Quora
- Gormanalysis
- Torch library documentation