Project-5

-Kirti Kshirsagar
- Saikiran Juttu

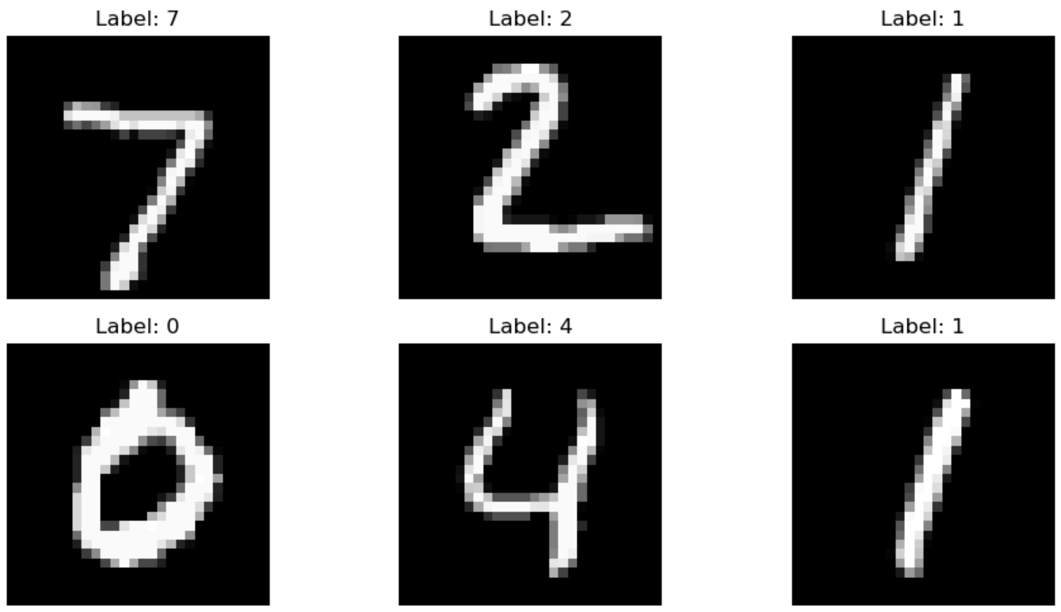
Project Description:

The project involves building, training, analyzing, and modifying a deep neural network for the recognition task of handwritten digits using the MNIST dataset. The project emphasizes using PyTorch for implementation due to its flexibility and efficiency. Initially, a convolutional neural network (CNN) architecture is constructed and trained to recognize digits. The network architecture consists of convolutional layers, max-pooling layers, dropout layers, and fully connected layers. After training, the network is saved for later use. Subsequent tasks involve analyzing the network's structure, evaluating its performance on test data, applying transfer learning to recognize Greek letters, and conducting experiments to optimize network architecture and training parameters.

Tasks:

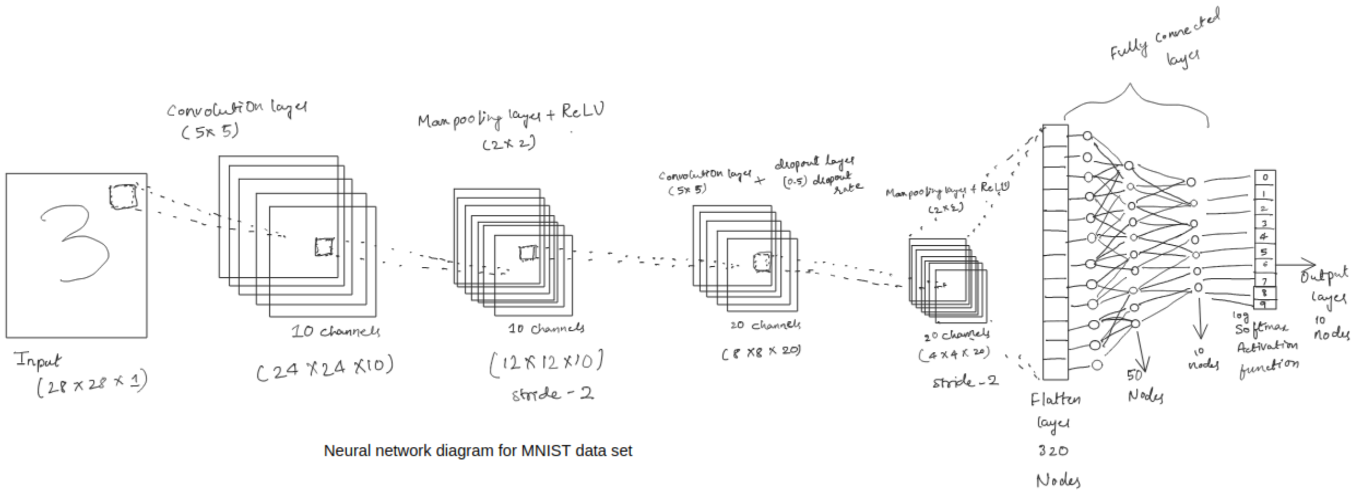
Task 1 -Build and train a network to recognize digits

A. In this task, we import the dataset and use matplotlib pyplot package or OpenCV to look at the first six example digits of the test set.



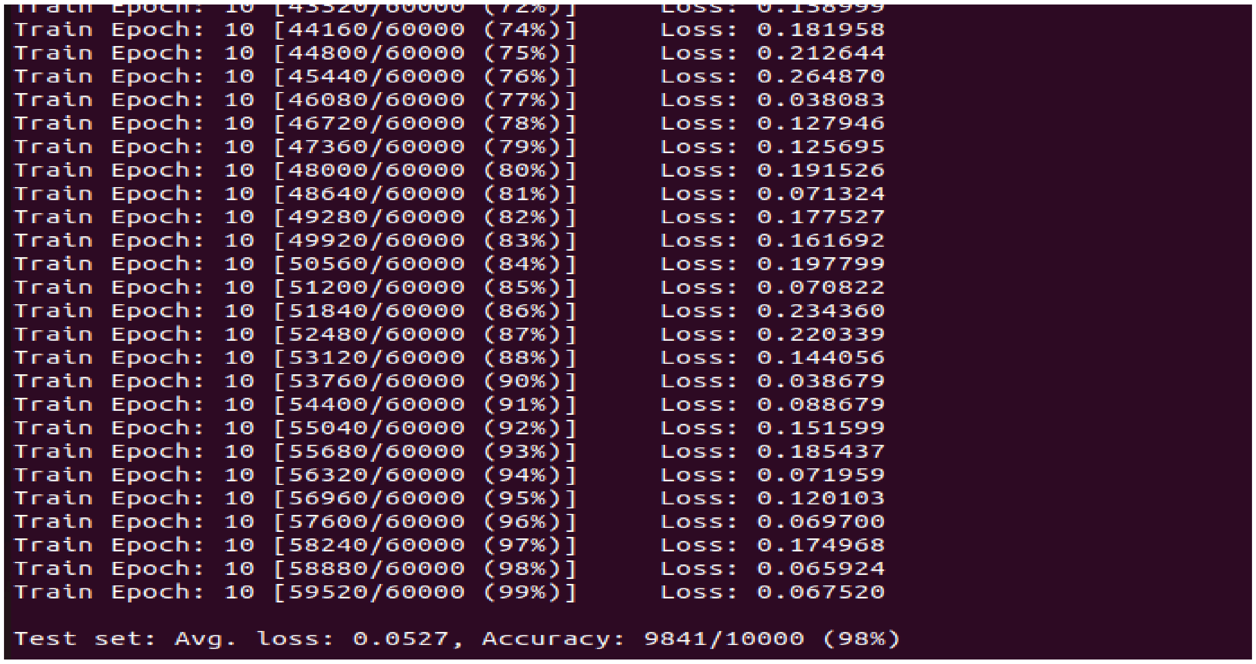
B. Then created a network with the following layers:

- A convolution layer with 10 5x5 filters
- A max pooling layer with a 2x2 window and a ReLU function applied.
- A convolution layer with 20 5x5 filters.
- A dropout layer with a 0.5 dropout rate (50%).
- A max pooling layer with a 2x2 window and a ReLU function applied.
- A flattening operation followed by a fully connected Linear layer with 50 nodes and a ReLU function on the output.
- A final fully connected Linear layer with 10 nodes and the log_softmax function applied to the output.

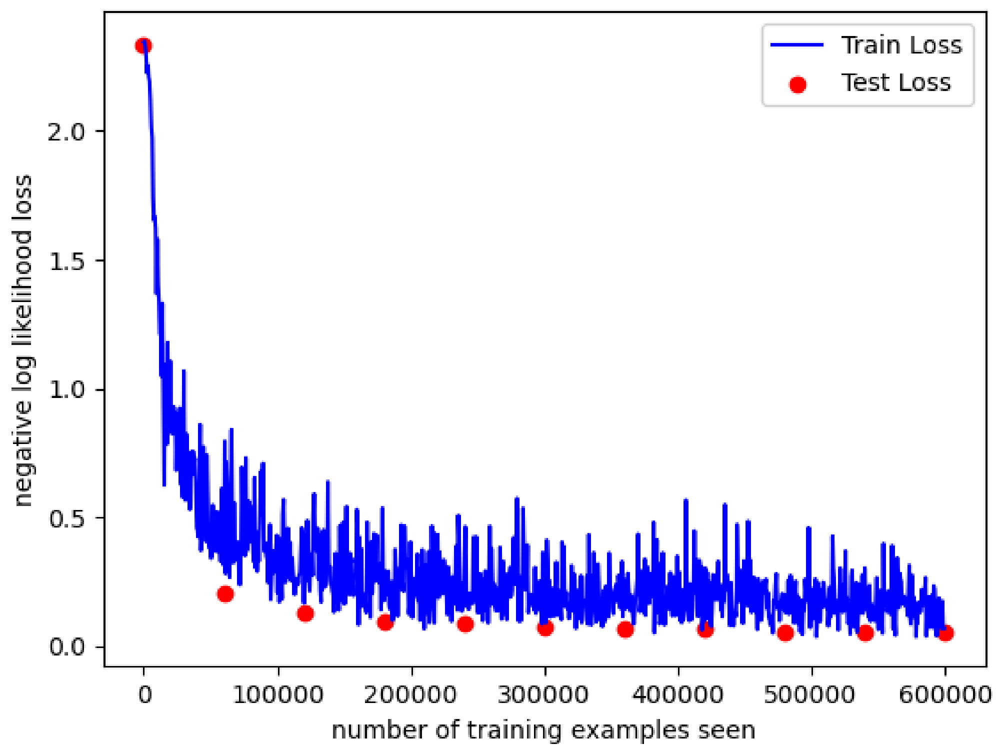


Neural network diagram for MNIST data set

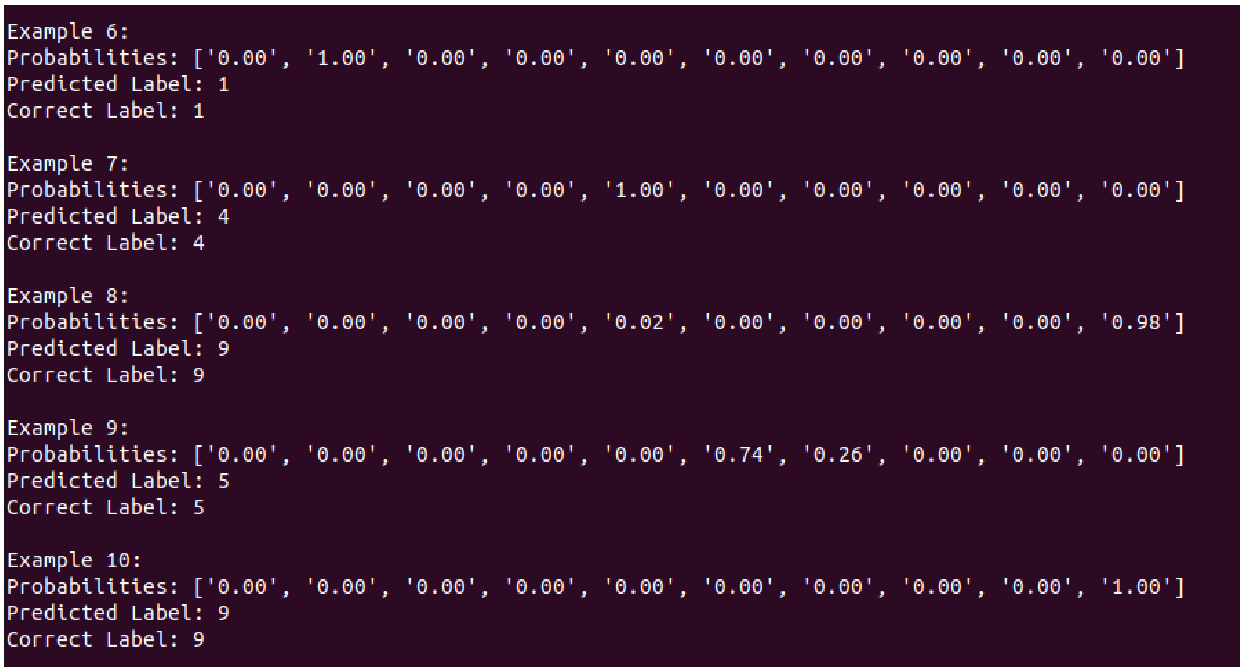
C. After creating a network we trained the model for 10 epochs and we evaluated the model on both the training and test sets after each epoch. We chose the batch size to be 64.



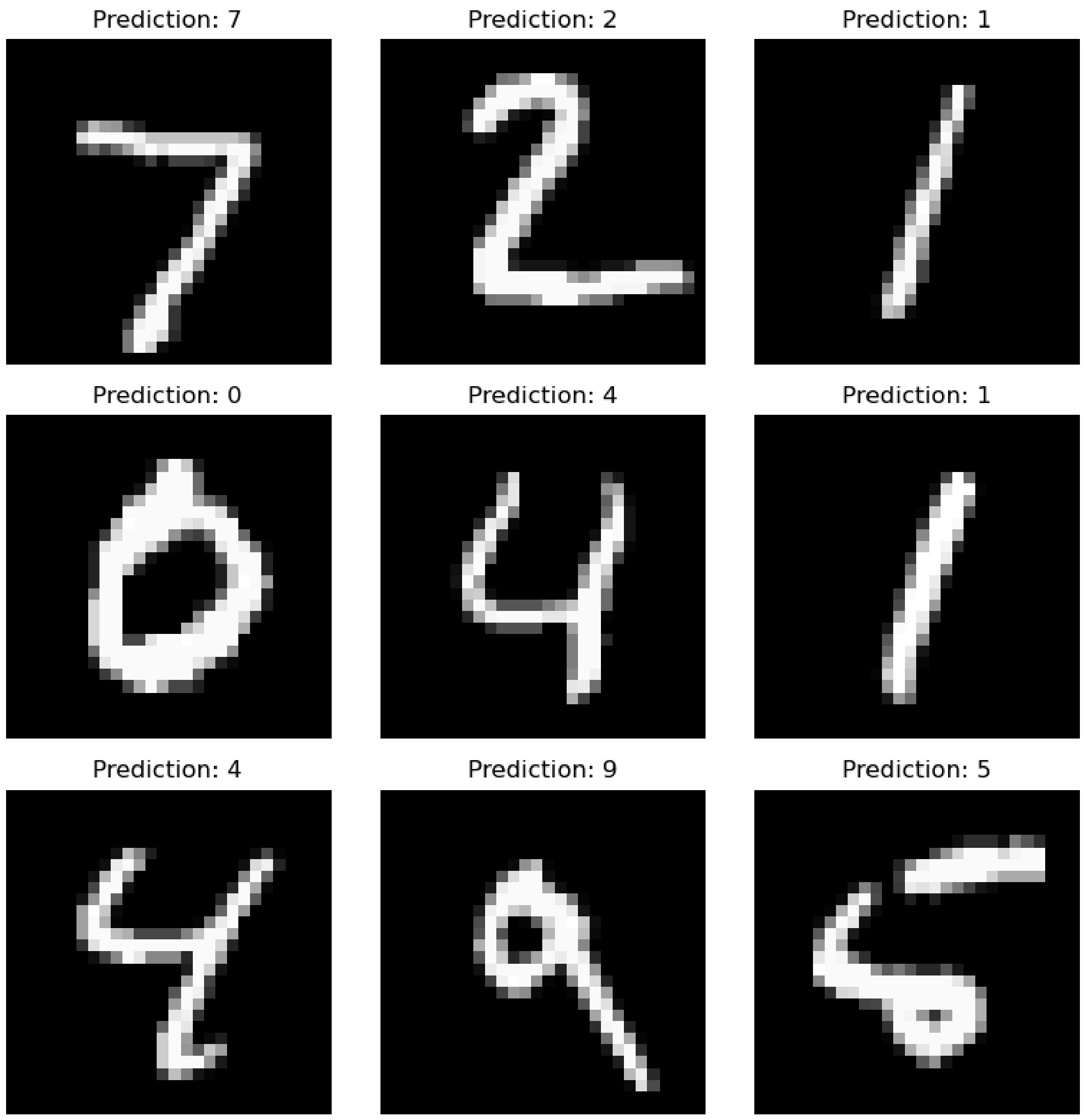
```
Train Epoch: 10 [43520/60000 (72%)]     Loss: 0.158999
Train Epoch: 10 [44160/60000 (74%)]     Loss: 0.181958
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.212644
Train Epoch: 10 [45440/60000 (76%)]     Loss: 0.264870
Train Epoch: 10 [46080/60000 (77%)]     Loss: 0.038083
Train Epoch: 10 [46720/60000 (78%)]     Loss: 0.127946
Train Epoch: 10 [47360/60000 (79%)]     Loss: 0.125695
Train Epoch: 10 [48000/60000 (80%)]     Loss: 0.191526
Train Epoch: 10 [48640/60000 (81%)]     Loss: 0.071324
Train Epoch: 10 [49280/60000 (82%)]     Loss: 0.177527
Train Epoch: 10 [49920/60000 (83%)]     Loss: 0.161692
Train Epoch: 10 [50560/60000 (84%)]     Loss: 0.197799
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.070822
Train Epoch: 10 [51840/60000 (86%)]     Loss: 0.234360
Train Epoch: 10 [52480/60000 (87%)]     Loss: 0.220339
Train Epoch: 10 [53120/60000 (88%)]     Loss: 0.144056
Train Epoch: 10 [53760/60000 (90%)]     Loss: 0.038679
Train Epoch: 10 [54400/60000 (91%)]     Loss: 0.088679
Train Epoch: 10 [55040/60000 (92%)]     Loss: 0.151599
Train Epoch: 10 [55680/60000 (93%)]     Loss: 0.185437
Train Epoch: 10 [56320/60000 (94%)]     Loss: 0.071959
Train Epoch: 10 [56960/60000 (95%)]     Loss: 0.120103
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.069700
Train Epoch: 10 [58240/60000 (97%)]     Loss: 0.174968
Train Epoch: 10 [58880/60000 (98%)]     Loss: 0.065924
Train Epoch: 10 [59520/60000 (99%)]     Loss: 0.067520

Test set: Avg. loss: 0.0527, Accuracy: 9841/10000 (98%)
```



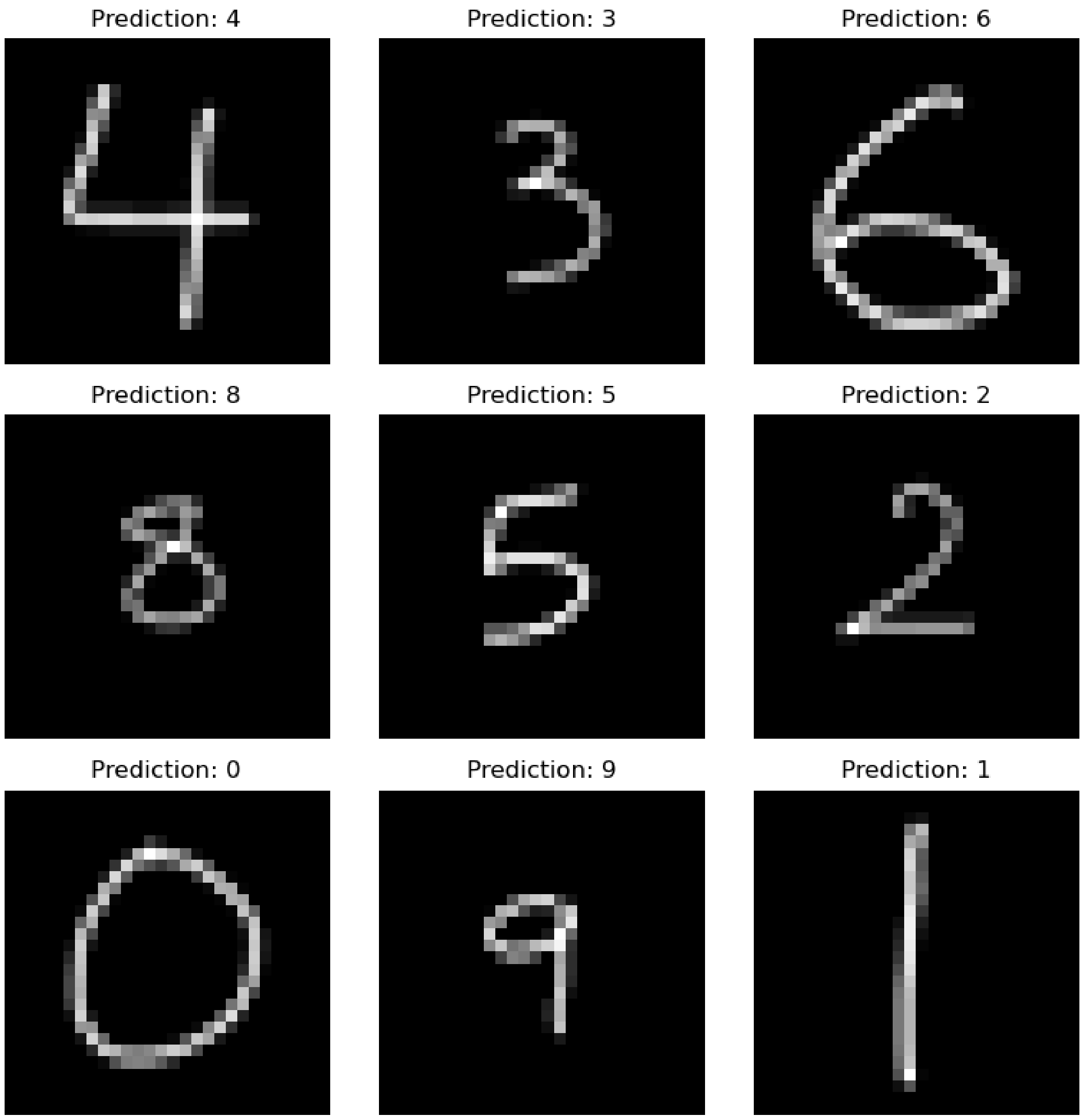
D. After training the network we saved it to a file in the results folder.
E. Here, In this task, we read the network in evaluation mode and run the model on the first 10 examples in the test set.



```
Example 6:
Probabilities: ['0.00', '1.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00']
Predicted Label: 1
Correct Label: 1

Example 7:
Probabilities: ['0.00', '0.00', '0.00', '0.00', '1.00', '0.00', '0.00', '0.00', '0.00', '0.00']
Predicted Label: 4
Correct Label: 4

Example 8:
Probabilities: ['0.00', '0.00', '0.00', '0.00', '0.02', '0.00', '0.00', '0.00', '0.00', '0.98']
Predicted Label: 9
Correct Label: 9

Example 9:
Probabilities: ['0.00', '0.00', '0.00', '0.00', '0.00', '0.74', '0.26', '0.00', '0.00', '0.00']
Predicted Label: 5
Correct Label: 5

Example 10:
Probabilities: ['0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '0.00', '1.00']
Predicted Label: 9
Correct Label: 9
```


Prediction: 7
Prediction: 2
Prediction: 1
Prediction: 0
Prediction: 4
Prediction: 1
Prediction: 4
Prediction: 9
Prediction: 5

F. In this task, we have created a folder containing handwritten digits and processed the images and matched their intensities with the intensities of the test data. We have then evaluated the model on these handwritten digits and it performed well.



Task 2-Examine your network

In this task, we examine our network and analyze how it processes the data. Here, we load the trained network as the first step and print the model. This shows us the structure of the network and the name of each layer.

```
Net(
  (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)
)
```

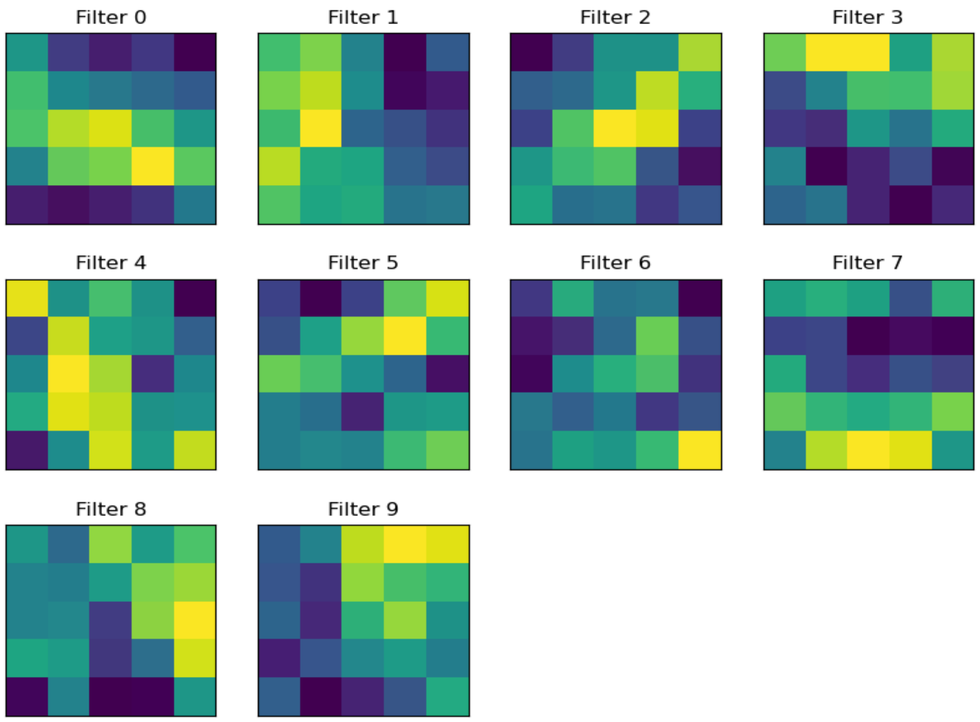
A. In this task, we analyze the first layer. We get the weights of the first layer.

```
Shape of the first layer weights: torch.Size([10, 1, 5, 5])
```

The result is a tensor that should have the shape [10, 1, 5, 5]. That means there are ten filters, 1 input channel, and each filter is 5x5 in size. We then print the filter weights and their shape.

```
Filter 0 shape: torch.Size([5, 5])
Filter 0 weights:
 tensor([[ 0.0713, -0.1836, -0.2589, -0.2005, -0.3590],
        [ 0.1705, -0.0030, -0.0009, -0.0543, -0.0870],
        [ 0.2033,  0.2647,  0.3077,  0.1447,  0.0883],
        [ 0.0244,  0.1929,  0.1976,  0.3466,  0.2189],
        [-0.2102, -0.2493, -0.1989, -0.1862, -0.0344]])

Filter 1 shape: torch.Size([5, 5])
Filter 1 weights:
 tensor([[ 0.1478,  0.2948,  0.0433, -0.4272, -0.1542],
        [ 0.2158,  0.3750,  0.0464, -0.4073, -0.3619],
        [ 0.1190,  0.4282, -0.1168, -0.1584, -0.2617],
        [ 0.3523,  0.0985,  0.1153, -0.1247, -0.1985],
        [ 0.2263,  0.0671,  0.1300, -0.0735, -0.0610]])

Filter 2 shape: torch.Size([5, 5])
Filter 2 weights:
 tensor([[-0.3662, -0.2106,  0.0857,  0.0660,  0.3306],
        [-0.0617, -0.0662,  0.0630,  0.3573,  0.1665],
        [-0.2017,  0.2173,  0.4004,  0.3745, -0.1746],
        [ 0.0463,  0.1586,  0.2083, -0.1507, -0.3078],
        [ 0.1480, -0.0055,  0.0383, -0.1667, -0.1220]])

Filter 3 shape: torch.Size([5, 5])
Filter 3 weights:
 tensor([[ 0.1740,  0.3036,  0.3363,  0.0478,  0.2781],
        [-0.1573, -0.0437,  0.1473,  0.1317,  0.2718],
        [-0.1784, -0.2136,  0.0585, -0.0558,  0.1322],
        [ 0.0149, -0.2758, -0.1892, -0.1128, -0.2451],
        [-0.0935, -0.0282, -0.1676, -0.2433, -0.1720]])
```

Like this all the filter weights are printed in our code

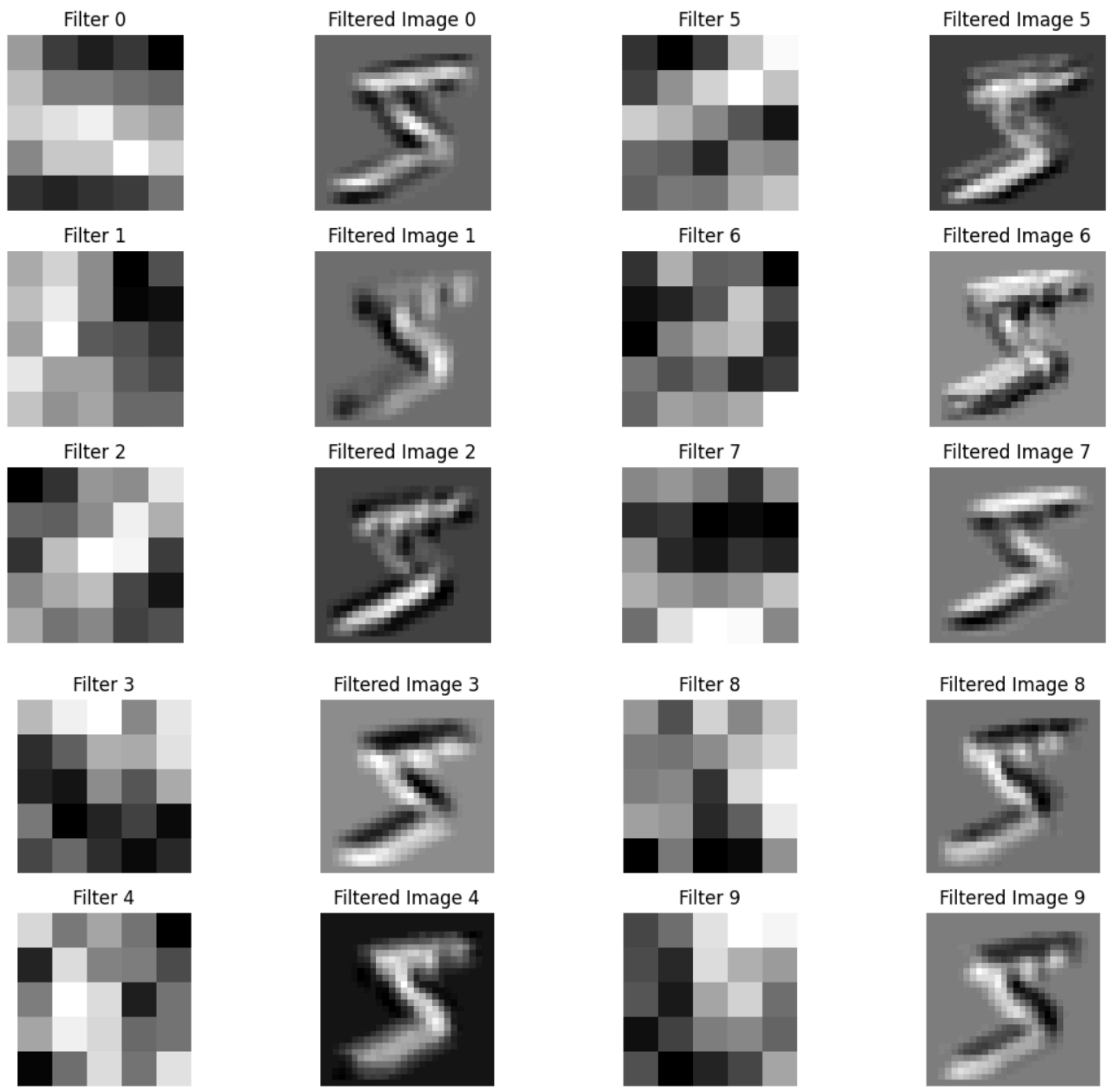


Visualizing the ten filters

B. In this task, we show the effect of the filters. Here, we use OpenCV's filter2D function to apply the 10 filters to the first training example image and generate a plot of the 10 filtered images.

   The intensity values in the following filter images represent the weights applied during the convolution operation. Positive values (shown as brighter areas) tend to activate the filter, while negative values (often darker areas) suppress the activation.

   In the context of our project, it makes sense for the filters to learn features that help distinguish between different digits. Edges are crucial for recognizing many digits (e.g., straight lines in ‘1’ or ‘7’, corners in ‘4’). Filters capturing textures or blobs could be beneficial for digits like ‘2’, ‘3’, ‘8’. Hence we can see how our result makes sense given the filters.

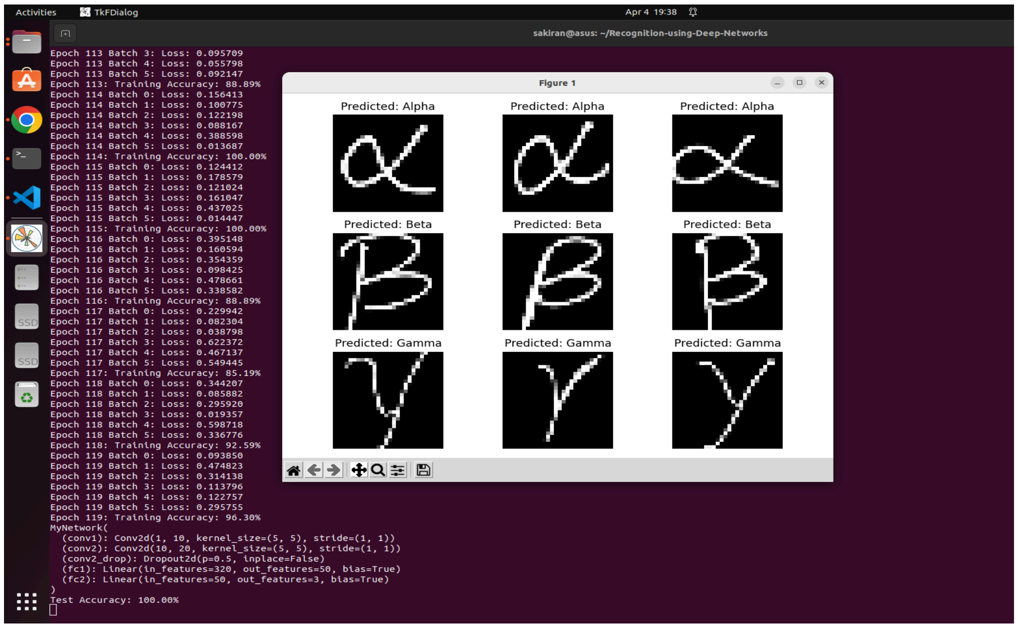


Task 3-Transfer Learning on Greek Letters

In this task, we used the previously developed MNIST digit recognition network to identify three specific Greek letters: alpha, beta, and gamma. To accomplish this, we utilized a dataset containing 27 samples of these Greek letters. First, we imported our code from task 1 to generate the MNIST network. Next, we loaded pre-trained weights from a file and froze the network weights. We then replaced the last layer with a new Linear layer consisting of three nodes. It's worth noting that the name of the last layer in the MNIST model from task 1 was displayed in our printout. We also included a code snippet demonstrating how to freeze all network weights.

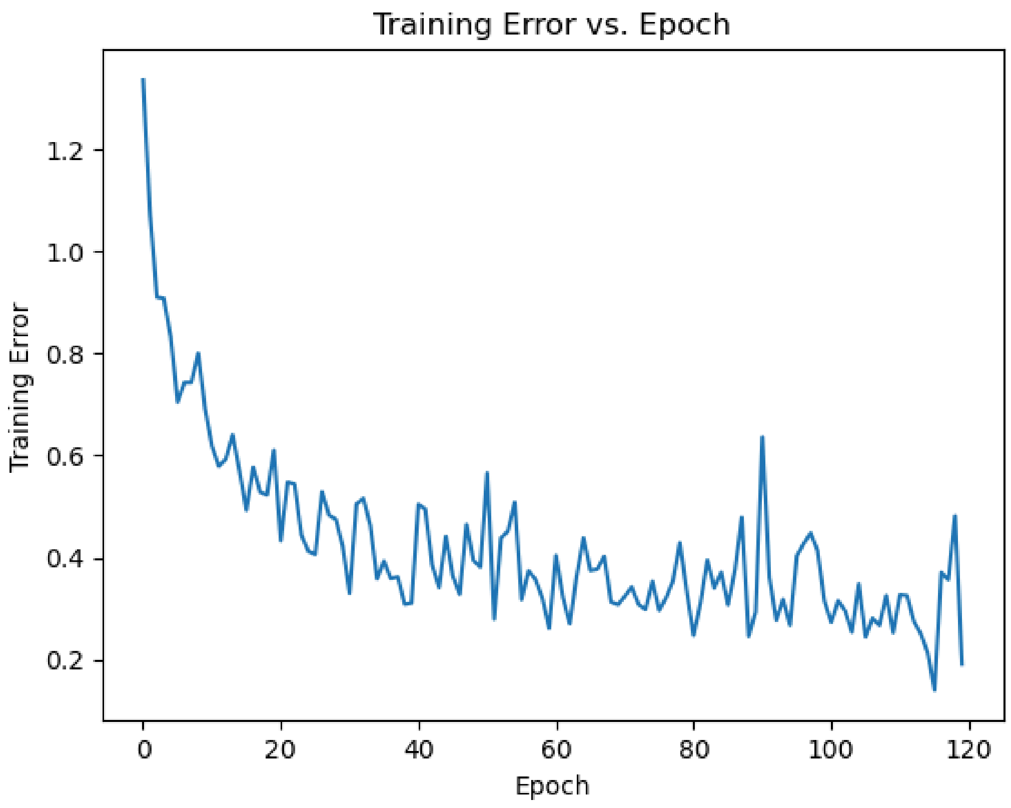
Subsequently, we investigated the number of epochs required to achieve nearly perfect identification using the 27 provided examples.

Upon experimenting with varying numbers of epochs, particularly exceeding 100 epochs, we achieved an impressive accuracy ranging between 96% to 98%.

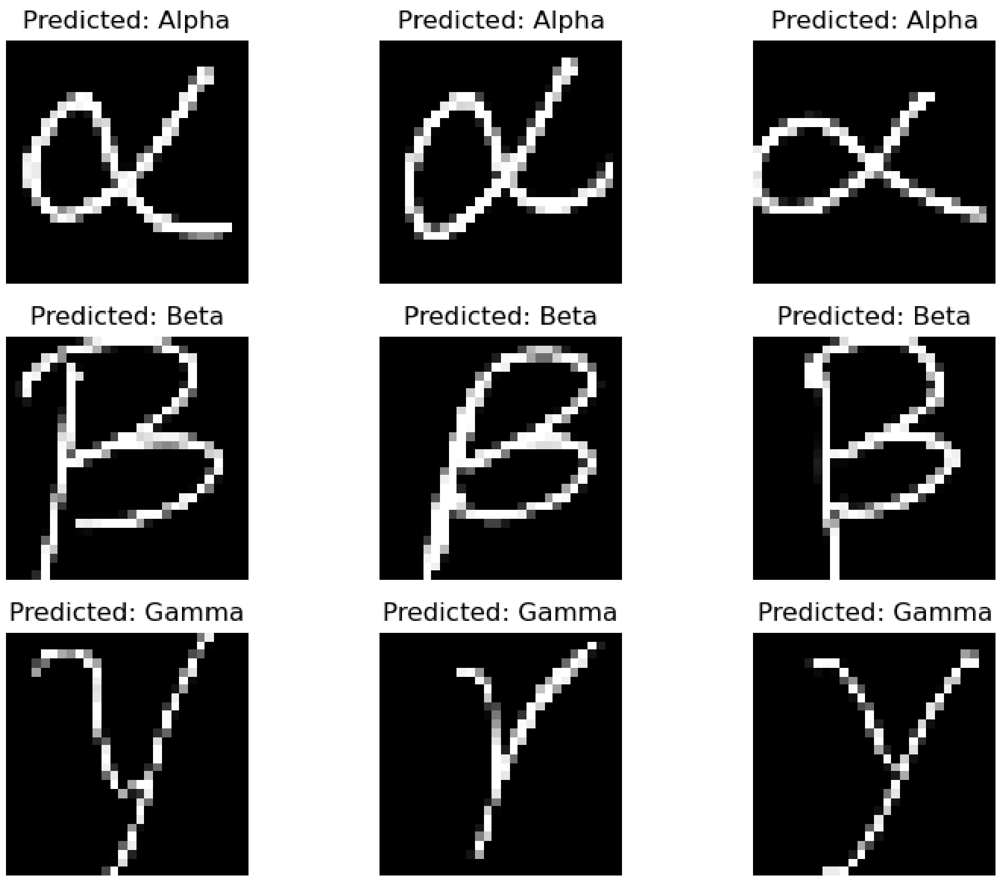


```
Epoch 113 Batch 3: Loss: 0.095709
Epoch 113 Batch 4: Loss: 0.055798
Epoch 113 Batch 5: Loss: 0.092147
Epoch 113: Training Accuracy: 88.89%
Epoch 114 Batch 0: Loss: 0.156413
Epoch 114 Batch 1: Loss: 0.100775
Epoch 114 Batch 2: Loss: 0.122198
Epoch 114 Batch 3: Loss: 0.088167
Epoch 114 Batch 4: Loss: 0.388598
Epoch 114 Batch 5: Loss: 0.013687
Epoch 114: Training Accuracy: 100.00%
Epoch 115 Batch 0: Loss: 0.124412
Epoch 115 Batch 1: Loss: 0.178579
Epoch 115 Batch 2: Loss: 0.121024
Epoch 115 Batch 3: Loss: 0.161047
Epoch 115 Batch 4: Loss: 0.437025
Epoch 115 Batch 5: Loss: 0.014447
Epoch 115: Training Accuracy: 100.00%
Epoch 116 Batch 0: Loss: 0.395148
Epoch 116 Batch 1: Loss: 0.160594
Epoch 116 Batch 2: Loss: 0.354359
Epoch 116 Batch 3: Loss: 0.098425
Epoch 116 Batch 4: Loss: 0.478661
Epoch 116 Batch 5: Loss: 0.338582
Epoch 116: Training Accuracy: 88.89%
Epoch 117 Batch 0: Loss: 0.229942
Epoch 117 Batch 1: Loss: 0.082304
Epoch 117 Batch 2: Loss: 0.038798
Epoch 117 Batch 3: Loss: 0.622372
Epoch 117 Batch 4: Loss: 0.467137
Epoch 117 Batch 5: Loss: 0.549445
Epoch 117: Training Accuracy: 85.19%
Epoch 118 Batch 0: Loss: 0.344207
Epoch 118 Batch 1: Loss: 0.085882
Epoch 118 Batch 2: Loss: 0.295920
Epoch 118 Batch 3: Loss: 0.019357
Epoch 118 Batch 4: Loss: 0.598718
Epoch 118 Batch 5: Loss: 0.336776
Epoch 118: Training Accuracy: 92.59%
Epoch 119 Batch 0: Loss: 0.093850
Epoch 119 Batch 1: Loss: 0.474823
Epoch 119 Batch 2: Loss: 0.314138
Epoch 119 Batch 3: Loss: 0.113796
Epoch 119 Batch 4: Loss: 0.122757
Epoch 119 Batch 5: Loss: 0.295755
Epoch 119: Training Accuracy: 96.30%
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)
)
Test Accuracy: 100.00%
```

The report also includes a plot depicting the training error, a printout of our modified network.



Later, we captured, cropped, and resized several alpha, beta, and gamma symbols to approximately 128x128 images to match the existing dataset. These images were then used to evaluate the classification accuracy of the trained network.
After evaluating the model on the provided images, we observed an accuracy of approximately 8-9 correct classifications out of 9 images.



Task 4-Design your own experiment (5 dimensions including extension 1)

A. **Plan:**
   In this task, we evaluated the effect of different network architecture parameters on the performance and training time of a deep neural network for the MNIST Fashion dataset classification task.
   The dimensions which we considered for our experiments are:
   - The size of the convolution filters
   - The dropout rates of the Dropout layer
   - Whether to add another dropout layer after the fully connected layer
   - The number of epochs of training
   - The batch size while training

These are the range of parameters which we used.
conv_filter_sizes_range = [(3, 3), (5, 5), (7, 7)]
epochs_range = [5, 6, 7]
dropout_rates_range = [0.1, 0.3, 0.5, 0.6]
add_dropout_fc_layer_range = [True, False]
batch_size_train_range = [32, 64]
We conducted 90 experiments exploring all the options in each of the above dimensions.We used nested for loops(Linear search strategy) to automate this process of iterating through a range of dimensions.

B. **Discussion on Hypotheses vs. Evaluation Results:**
Convolution Filter Size:
- **Hypothesis:** Larger filter sizes might capture more complex patterns but may increase computational cost. Smaller filter sizes might capture finer details but may lead to overfitting.
- **Evaluation:** Filter sizes of 5x5 and 7x7 consistently appeared in the top-performing combinations, suggesting that larger filter sizes indeed contributed to better test accuracies without significantly increasing training time.

Epochs:
- **Hypothesis:** Increasing epochs might improve model convergence and accuracy up to a certain point, but too many epochs could lead to overfitting.
- **Evaluation:** The choice of the number of epochs varied across the top-performing combinations, indicating that the optimal number of epochs depended on other hyperparameters. However, longer training times were associated with higher epoch counts, suggesting a trade-off between training time and model performance.

Dropout Rate:
- **Hypothesis:** Higher dropout rates might provide stronger regularization, preventing overfitting but potentially hindering learning if too aggressive.
- **Evaluation:** Dropout rates of 0.1 and None consistently appeared in the top-performing combinations. The highest dropout rate of 0.6 generally led to lower test accuracies, suggesting that it might have been too aggressive in regularizing the model.

Training Batch Size:

- **Hypothesis:** Larger batch sizes might lead to faster convergence due to more stable updates, but they may require more memory and computational resources.
- **Evaluation:** Batch sizes of 32 consistently appeared in the top-performing combinations, indicating that smaller batch sizes might lead to better generalization. However, the difference in training time between batch sizes of 32 and 64 was not significant in most cases.

Add Dropout FC Layer:

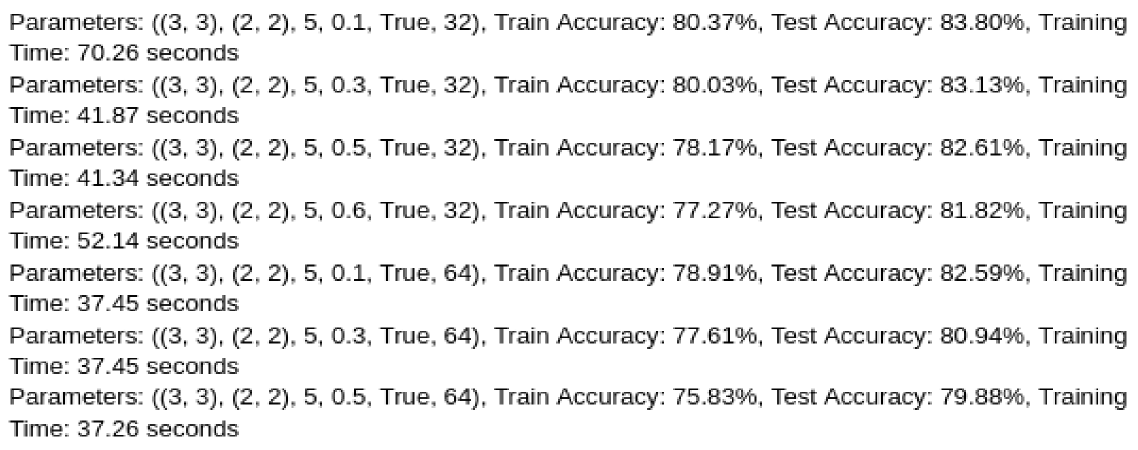
- **Hypothesis:** Adding a dropout layer after the fully connected layer may improve generalization, especially if the model is prone to memorizing the training data.
- **Evaluation:** The absence of a dropout layer after the fully connected layer consistently led to top combinations with better test accuracy. Additionally, it might have contributed to lower training times compared to configurations with dropout layers.

Training Time Analysis:

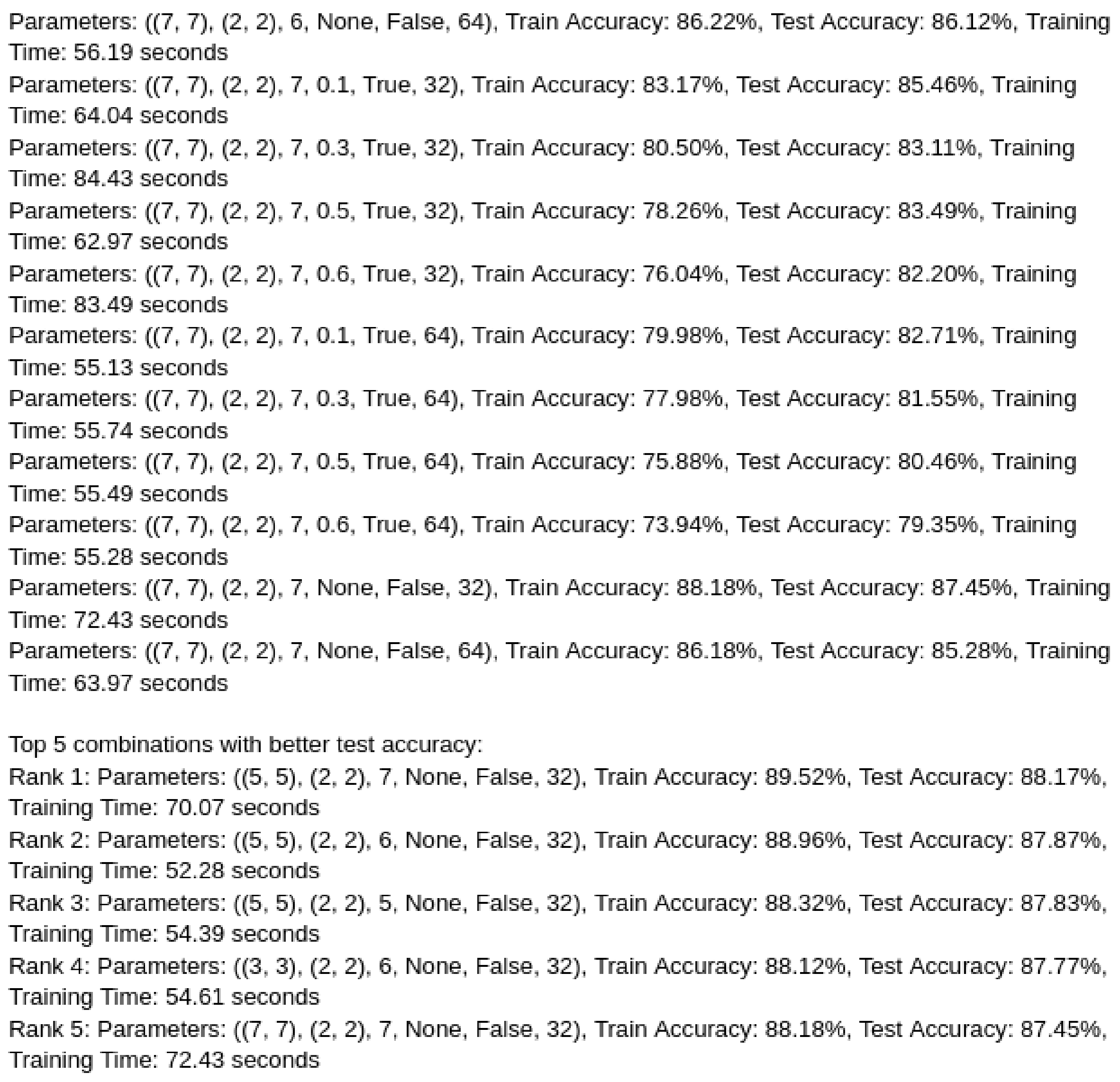
- Parameters: ((3, 3), (2, 2), 5, 0.3, True, 64), Train Accuracy: 77.61%, Test Accuracy: 80.94%, Training Time: 37.45 seconds
- This combination had a training time of 37.45 seconds and achieved a test accuracy of 80.94%. It used a 3x3 convolution filter, a 2x2 pooling layer, 5 epochs, a dropout rate of 0.3, with batch normalization, and a batch size of 64.
- It's important to note that the training time can vary depending on factors such as the complexity of the model, hardware specifications, and software optimizations. In this case, the mentioned combination had the least training time among the evaluated configurations.

C. **Overall Conclusion:**

The evaluation results generally supported the hypotheses regarding dropout rates, batch sizes, convolution filter sizes, and epochs. Additionally, the trade-off between model performance and training time was evident, with certain hyperparameter configurations leading to better test accuracies with reasonable training times. The top-performing combinations typically included moderate dropout rates, smaller batch sizes, larger convolution filter sizes, and appropriate numbers of epochs.



```
Parameters: ((3, 3), (2, 2), 5, 0.1, True, 32), Train Accuracy: 80.37%, Test Accuracy: 83.80%, Training
Time: 70.26 seconds
Parameters: ((3, 3), (2, 2), 5, 0.3, True, 32), Train Accuracy: 80.03%, Test Accuracy: 83.13%, Training
Time: 41.87 seconds
Parameters: ((3, 3), (2, 2), 5, 0.5, True, 32), Train Accuracy: 78.17%, Test Accuracy: 82.61%, Training
Time: 41.34 seconds
Parameters: ((3, 3), (2, 2), 5, 0.6, True, 32), Train Accuracy: 77.27%, Test Accuracy: 81.82%, Training
Time: 52.14 seconds
Parameters: ((3, 3), (2, 2), 5, 0.1, True, 64), Train Accuracy: 78.91%, Test Accuracy: 82.59%, Training
Time: 37.45 seconds
Parameters: ((3, 3), (2, 2), 5, 0.3, True, 64), Train Accuracy: 77.61%, Test Accuracy: 80.94%, Training
Time: 37.45 seconds
Parameters: ((3, 3), (2, 2), 5, 0.5, True, 64), Train Accuracy: 75.83%, Test Accuracy: 79.88%, Training
Time: 37.26 seconds
```

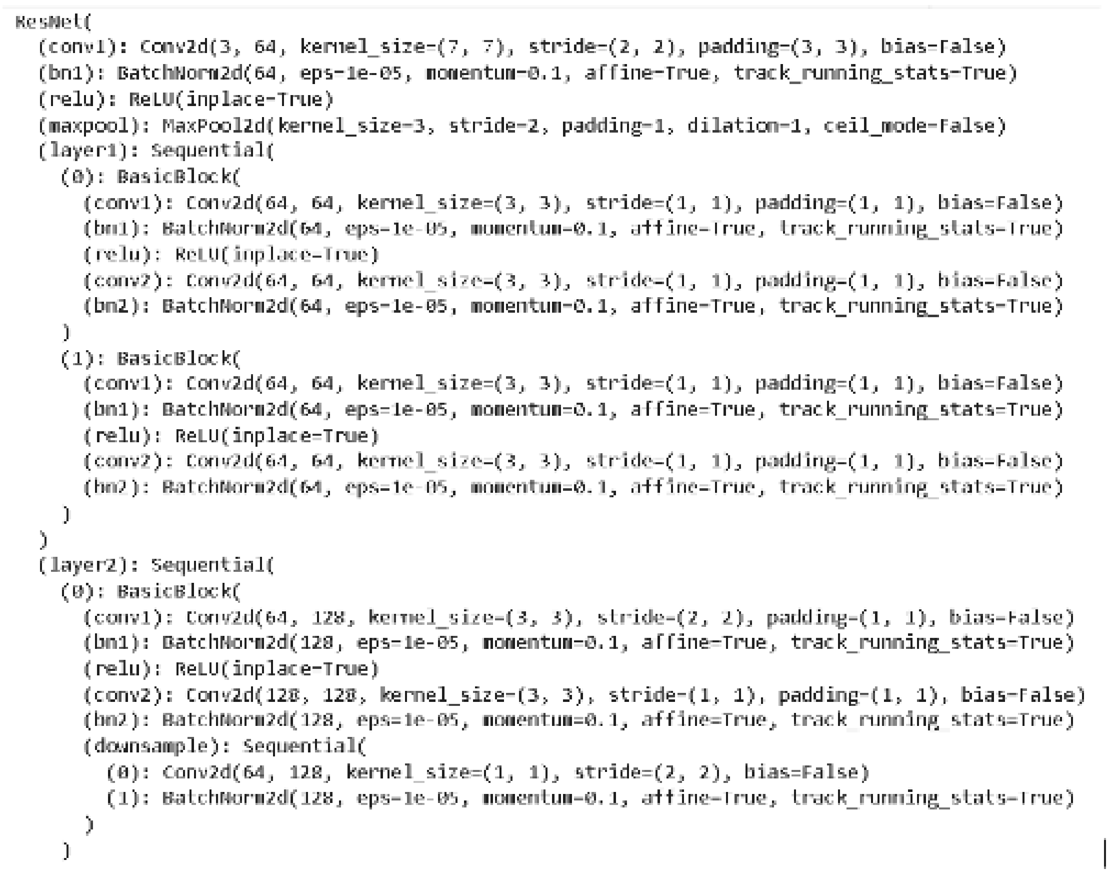


```
Parameters: ((7, 7), (2, 2), 6, None, False, 64), Train Accuracy: 86.22%, Test Accuracy: 86.12%, Training
Time: 56.19 seconds
Parameters: ((7, 7), (2, 2), 7, 0.1, True, 32), Train Accuracy: 83.17%, Test Accuracy: 85.46%, Training
Time: 64.04 seconds
Parameters: ((7, 7), (2, 2), 7, 0.3, True, 32), Train Accuracy: 80.50%, Test Accuracy: 83.11%, Training
Time: 84.43 seconds
Parameters: ((7, 7), (2, 2), 7, 0.5, True, 32), Train Accuracy: 78.26%, Test Accuracy: 83.49%, Training
Time: 62.97 seconds
Parameters: ((7, 7), (2, 2), 7, 0.6, True, 32), Train Accuracy: 76.04%, Test Accuracy: 82.20%, Training
Time: 83.49 seconds
Parameters: ((7, 7), (2, 2), 7, 0.1, True, 64), Train Accuracy: 79.98%, Test Accuracy: 82.71%, Training
Time: 55.13 seconds
Parameters: ((7, 7), (2, 2), 7, 0.3, True, 64), Train Accuracy: 77.98%, Test Accuracy: 81.55%, Training
Time: 55.74 seconds
Parameters: ((7, 7), (2, 2), 7, 0.5, True, 64), Train Accuracy: 75.88%, Test Accuracy: 80.46%, Training
Time: 55.49 seconds
Parameters: ((7, 7), (2, 2), 7, 0.6, True, 64), Train Accuracy: 73.94%, Test Accuracy: 79.35%, Training
Time: 55.28 seconds
Parameters: ((7, 7), (2, 2), 7, None, False, 32), Train Accuracy: 88.18%, Test Accuracy: 87.45%, Training
Time: 72.43 seconds
Parameters: ((7, 7), (2, 2), 7, None, False, 64), Train Accuracy: 86.18%, Test Accuracy: 85.28%, Training
Time: 63.97 seconds

Top 5 combinations with better test accuracy:
Rank 1: Parameters: ((5, 5), (2, 2), 7, None, False, 32), Train Accuracy: 89.52%, Test Accuracy: 88.17%,
Training Time: 70.07 seconds
Rank 2: Parameters: ((5, 5), (2, 2), 6, None, False, 32), Train Accuracy: 88.96%, Test Accuracy: 87.87%,
Training Time: 52.28 seconds
Rank 3: Parameters: ((5, 5), (2, 2), 5, None, False, 32), Train Accuracy: 88.32%, Test Accuracy: 87.83%,
Training Time: 54.39 seconds
Rank 4: Parameters: ((3, 3), (2, 2), 6, None, False, 32), Train Accuracy: 88.12%, Test Accuracy: 87.77%,
Training Time: 54.61 seconds
Rank 5: Parameters: ((7, 7), (2, 2), 7, None, False, 32), Train Accuracy: 88.18%, Test Accuracy: 87.45%,
Training Time: 72.43 seconds
```

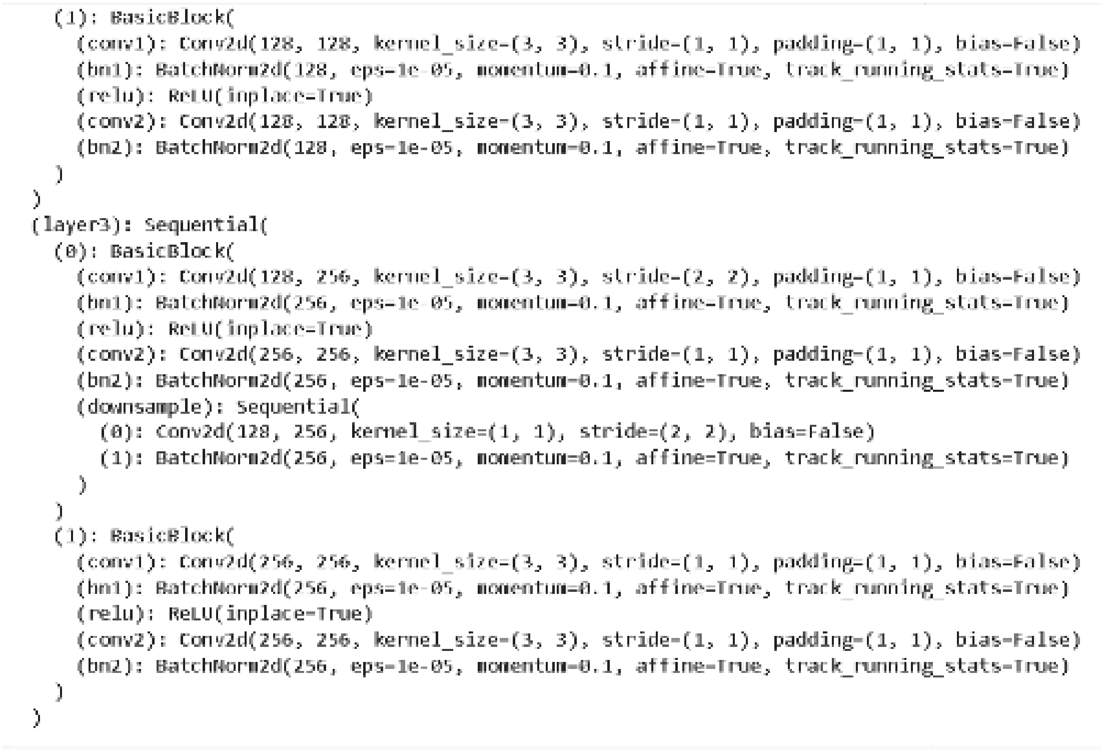
Results of the experiments

Extension:

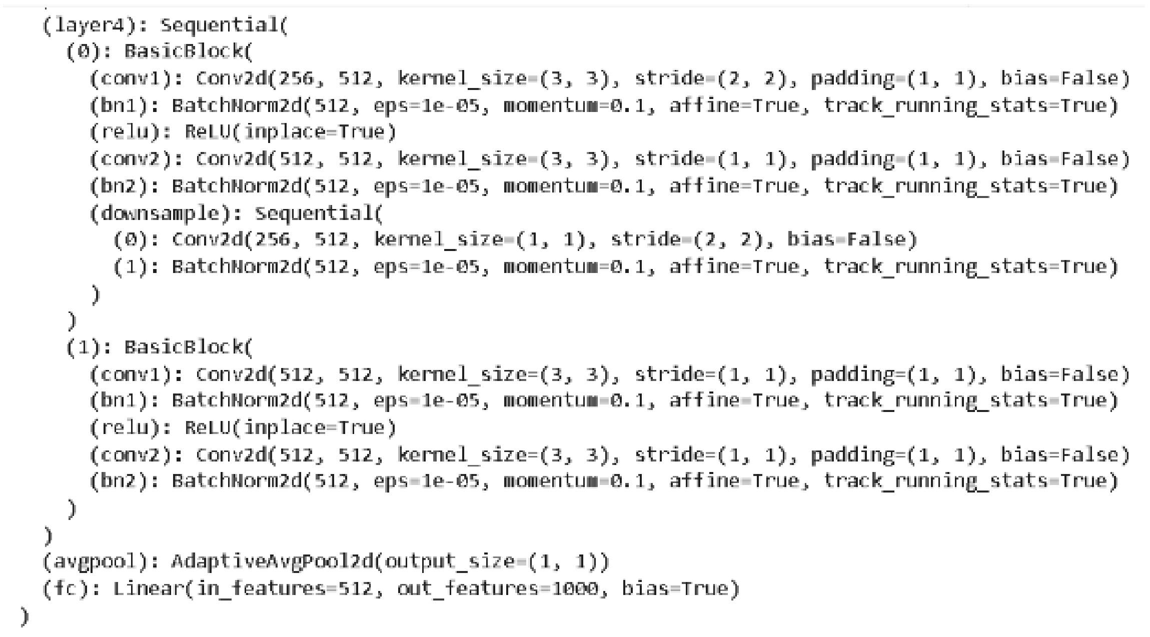
1. We Evaluated more dimensions on task 4. Further details are mentioned in task 4.
2. We implemented the extension in which we used a pre-trained network such as ResNet18, loaded it and evaluated its first couple of convolutional layers as in task 2. Below is what we learned:



```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
```



```
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```



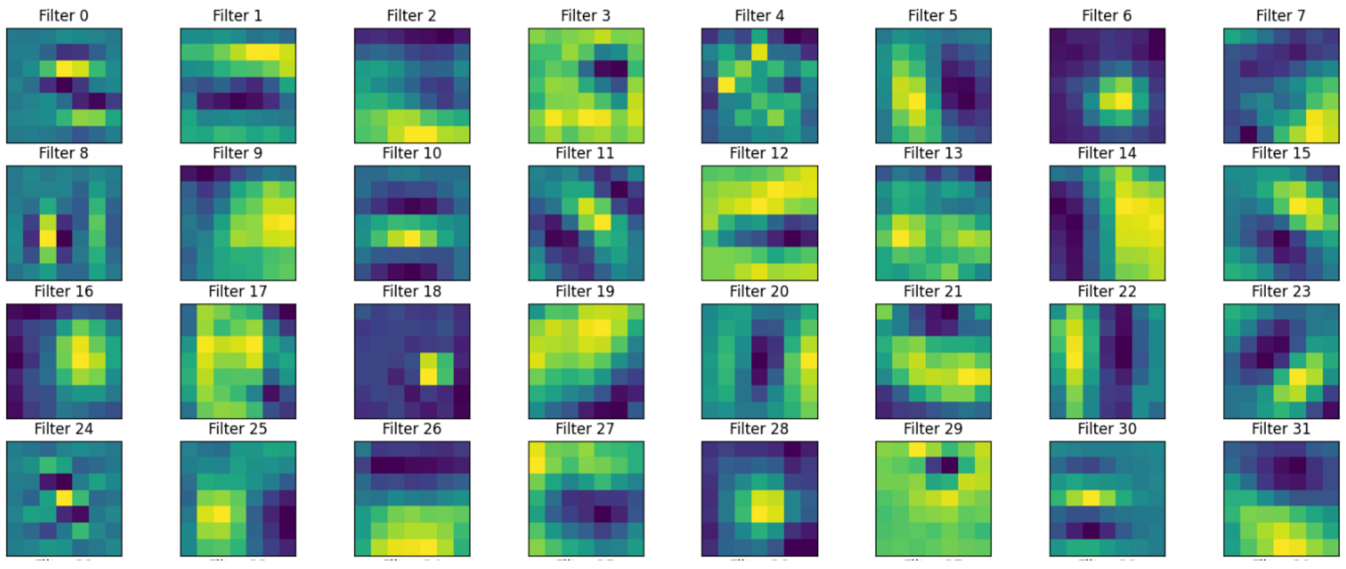
```
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

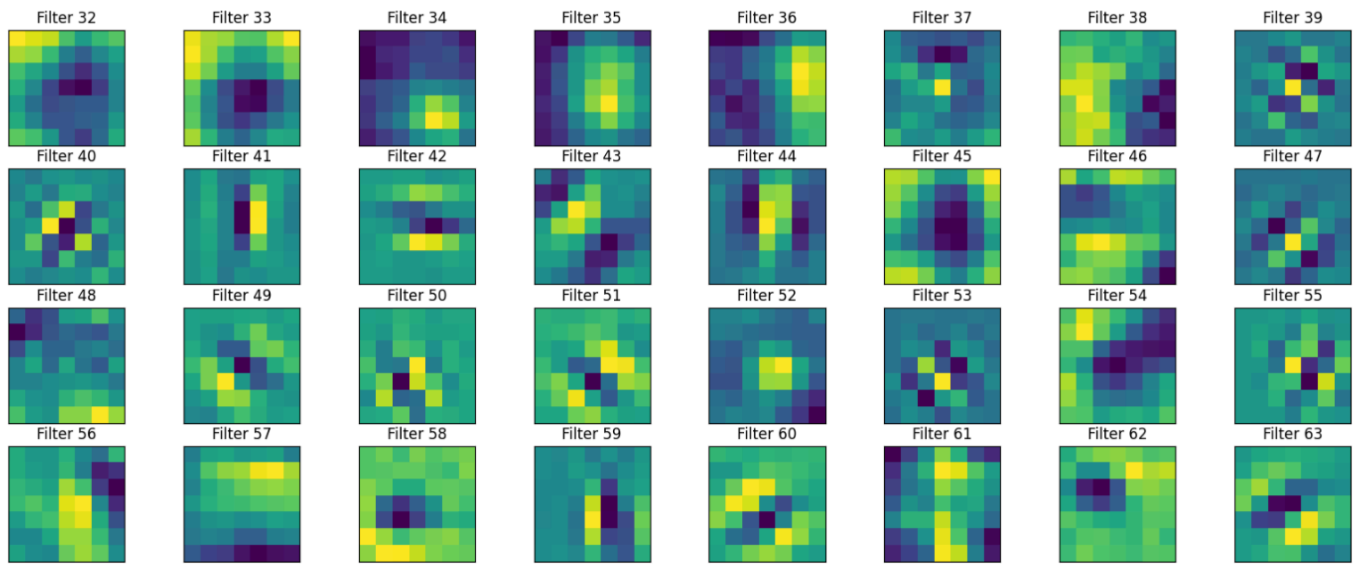
Structure of the pre-trained model

When we analyzed the first and the second layer of convoluted network, we got the following weights:

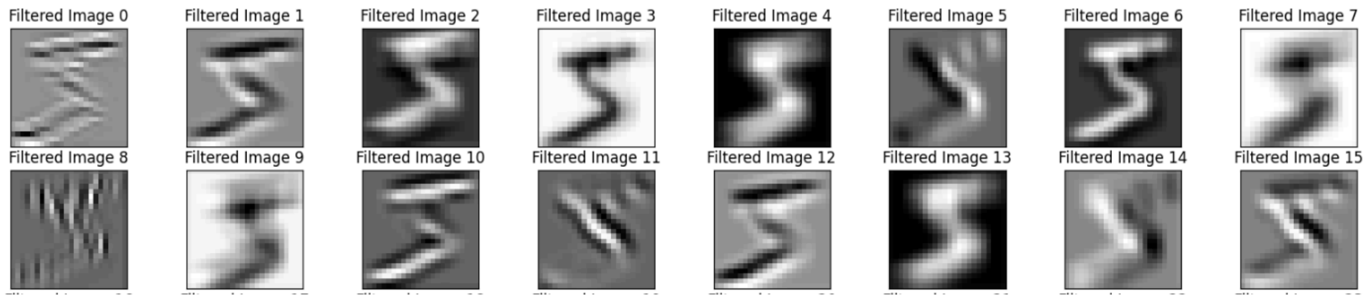
```
Shape of the first layer weights: torch.Size([64, 3, 7, 7])
Shape of the second layer weights: torch.Size([64, 64, 3, 3])
```

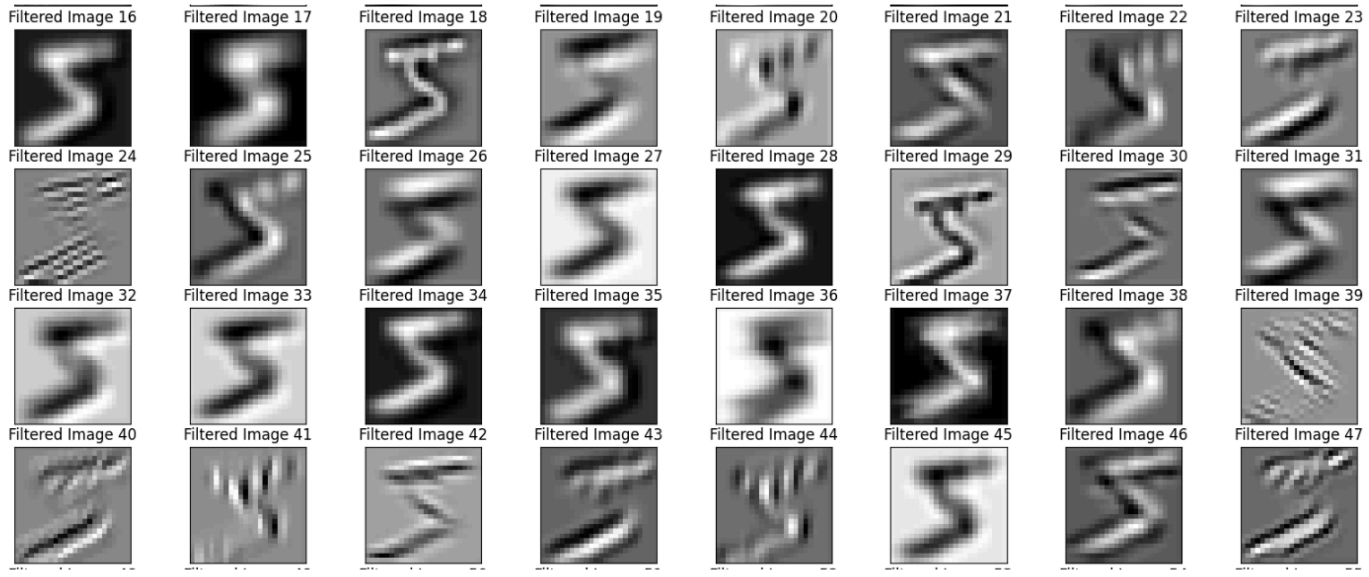
We then visualized the 64 filters present in the first layer:

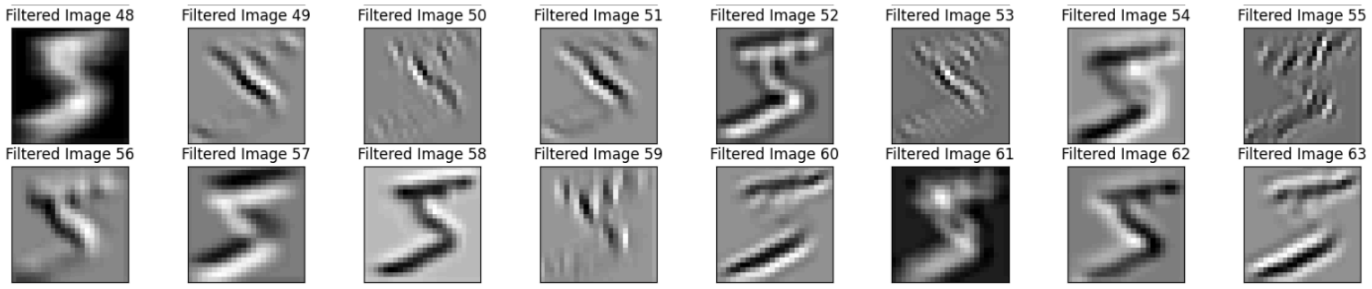




Using OpenCV's filter2D function, we showed the effect of these 64 filters on the image from the training set:



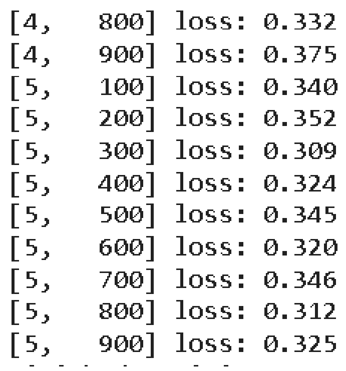




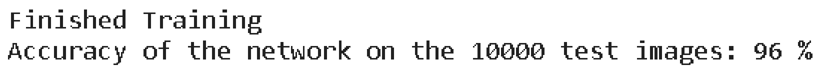
As stated from task-2, the intensity values in these filter images represent the weights applied during the convolution operation. Positive values (shown as brighter areas) tend to activate the filter, while negative values (often darker areas) suppress the activation.

3. For the third extension, we replaced the first layer of the MNIST network with Gabor filters and retrained the rest of the network, holding the first layer constant. So, when I initially defined the hyper-parameters of this new network, the model performed very poorly, I received an accuracy of 20%. But later after a lot of trial and error, when I set the number of epochs as 5 and the number of gabor filters as 2, I achieved super good accuracy of about 96%. So this network did perform almost as good as our original network which gives accurate results about 98% of the time.

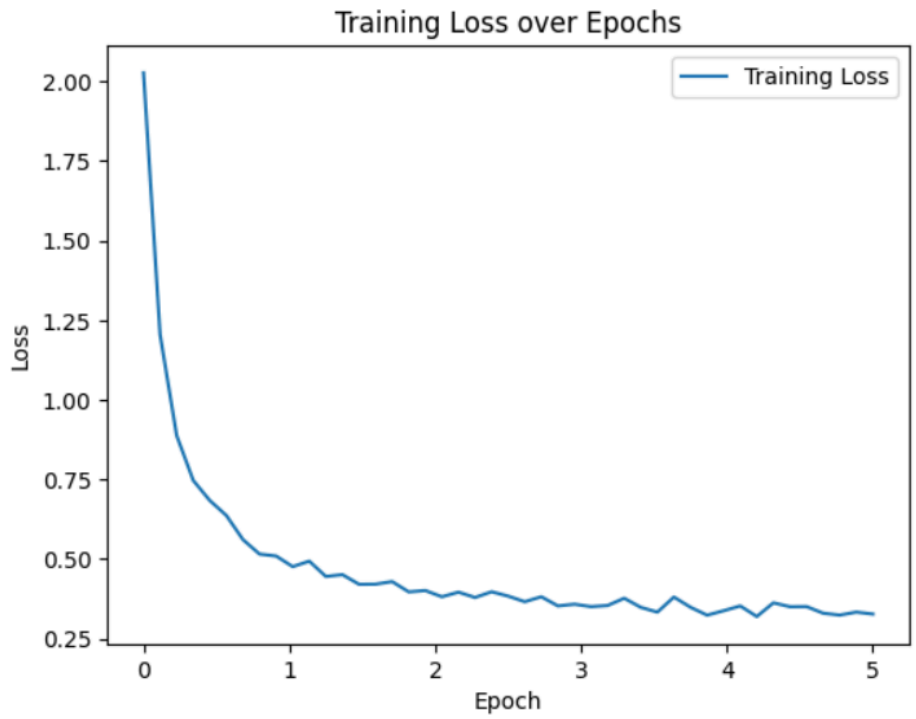
```
[1,   100] loss: 1.974
[1,   200] loss: 1.168
[1,   300] loss: 0.915
[1,   400] loss: 0.777
[1,   500] loss: 0.706
[1,   600] loss: 0.634
[1,   700] loss: 0.586
[1,   800] loss: 0.551
[1,   900] loss: 0.510
[2,   100] loss: 0.481
[2,   200] loss: 0.483
[2,   300] loss: 0.486
[2,   400] loss: 0.455
[2,   500] loss: 0.415
[2,   600] loss: 0.441
[2,   700] loss: 0.435
[2,   800] loss: 0.419
[2,   900] loss: 0.415
[3,   100] loss: 0.390
[3,   200] loss: 0.406
[3,   300] loss: 0.385
[3,   400] loss: 0.375
[3,   500] loss: 0.391
[3,   600] loss: 0.393
[3,   700] loss: 0.367
[3,   800] loss: 0.384
[3,   900] loss: 0.365
[4,   100] loss: 0.362
[4,   200] loss: 0.354
[4,   300] loss: 0.339
[4,   400] loss: 0.342
[4,   500] loss: 0.359
[4,   600] loss: 0.361
[4,   700] loss: 0.358
```



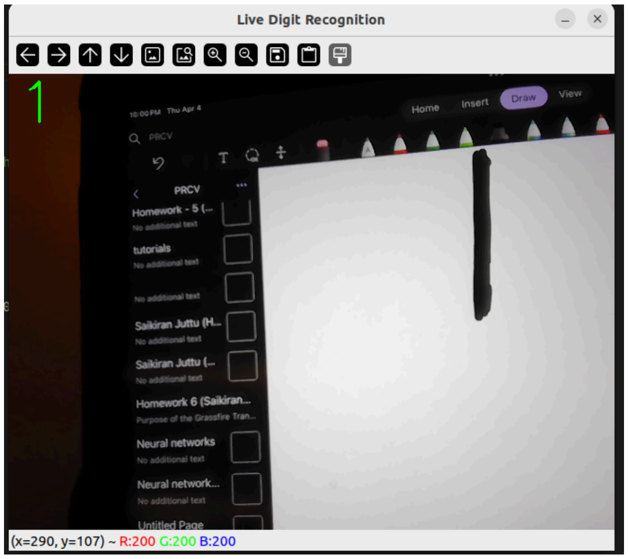
```
[4,   800] loss: 0.332
[4,   900] loss: 0.375
[5,   100] loss: 0.340
[5,   200] loss: 0.352
[5,   300] loss: 0.309
[5,   400] loss: 0.324
[5,   500] loss: 0.345
[5,   600] loss: 0.320
[5,   700] loss: 0.346
[5,   800] loss: 0.312
[5,   900] loss: 0.325
```

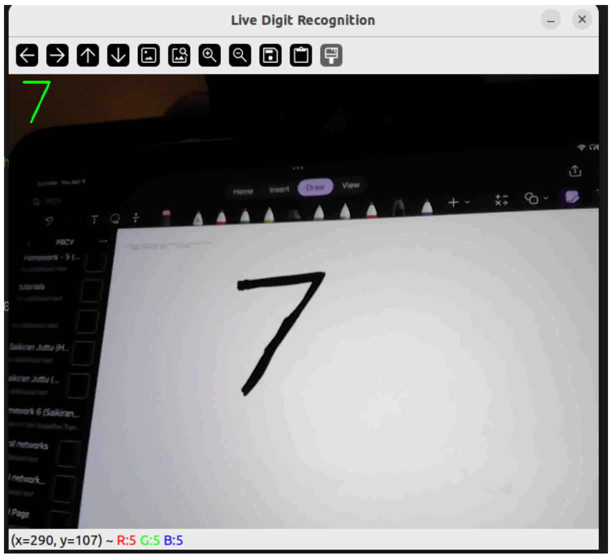


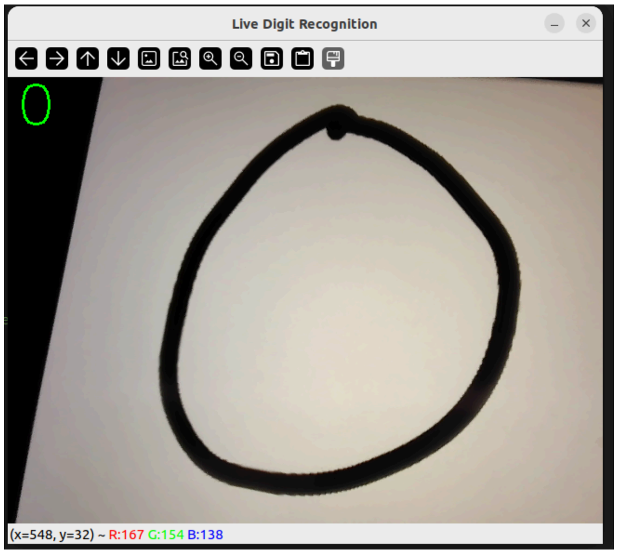
```
Finished Training
Accuracy of the network on the 10000 test images: 96 %
```

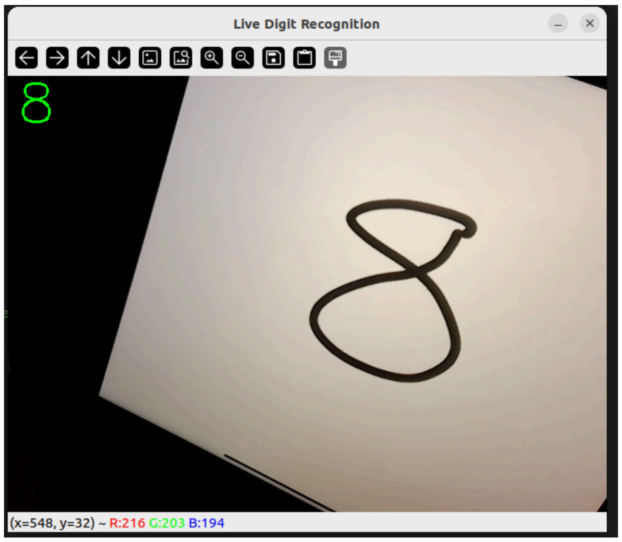


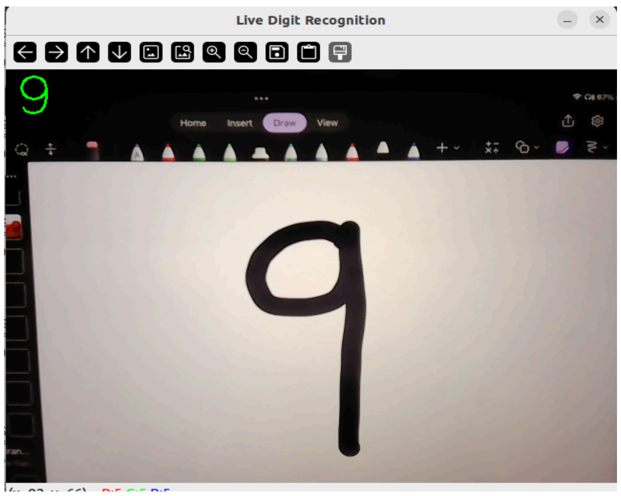
4. For the live video digit recognition system, we utilized a trained neural network model called MyNetwork, which was previously trained on the MNIST dataset for digit classification. This model, along with its corresponding optimizer, was loaded from saved checkpoint files (model.pth and optimizer.pth). With OpenCV, we initialized the webcam to capture video frames in real-time. Each frame underwent preprocessing to ensure compatibility with the neural network's input format, involving resizing, converting to grayscale, and normalizing pixel values. Subsequently, the preprocessed frame was passed through the trained neural network to obtain predictions for the digit depicted in the frame. Finally, the live video stream was displayed alongside the predicted digit, providing real-time digit recognition functionality. This system demonstrates the practical application of deep learning techniques in computer vision for real-time tasks.

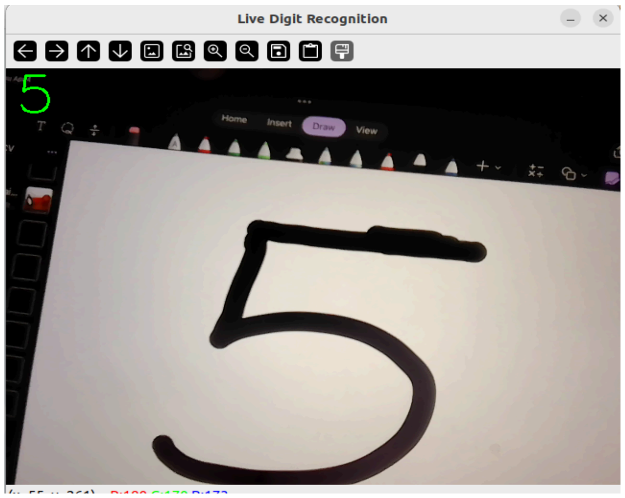












These are the results of the live digit recognition, sometimes it is not so accurate because of higher frame rate.

Reflection:

This project provided valuable insights into deep learning for recognition tasks. We learned about:

- Convolutional Neural Networks (CNNs): Their structure, layers (convolution, pooling, fully connected), and training process.
- MNIST Dataset: Importance of a well-defined dataset for training and evaluating recognition models.
- Transfer Learning: Reusing a pre-trained network for a new task by adapting its final layers.
- Hyperparameter Tuning: Exploring different network configurations to optimize performance.
- Importance of Experimentation: Understanding how network architecture influences recognition accuracy and training time.

Overall, the project provided a practical experience in building, training, and analyzing deep networks for image recognition.

Acknowledgement of material consulted:

- OpenCV Documentation
- Stack Overflow
- Towards Data Science
- Tutorial
- Neural Network pytorch

Installations:

So we are working on Windows 10, VScode, OpenCV 4.9(recent one), Pytorch and python.