Project 5: Recognition using Deep Networks

Hardik Devrangadi devrangadi.h@northeastern.edu
Thejaswini Goriparthi goriparthi.t@northeastern.edu

Recognition using Deep Networks on MNIST Datasets and Custom Datasets

This project involved using deep neural networks to recognize digits from the MNIST dataset and Greek letters from a custom dataset. First, a neural network was built and trained using the pyTorch package and torchvision to recognize digits from the MNIST dataset. Then, transfer learning was applied to the same network to recognize Greek letters from the custom dataset. Additionally, different hyperparameters for the FashionMNIST dataset were explored to identify the best model that yielded the highest accuracy on the test set. Lastly, the first few convolution layers of a pre-trained AlexNet model were evaluated and the effect of 2D filters on the CIFAR10 dataset was visualized. The pyTorch package and torchvision were used to achieve these goals.

Tasks

Task 1: Build and Train a Network to Recognize Digits

In this Task, we build and train the network to recognize the digits from the MNIST digit dataset

A. Get the MNIST digit data set

To begin with, the MNIST digit dataset was obtained for the task of recognizing digits. This dataset contains 60,000 training and 10,000 testing images of 28x28 pixels, representing handwritten digits from 0 to 9.
To use the MNIST dataset for training and testing the neural network, a dataloader was used. In this case, the dataloader normalized the pixel values of the images to be between 0 and 1. This helped in improving the performance of the network by reducing the scale of the input values.

Figure 1 shows a plot of the first six example digits of the MINST dataset.

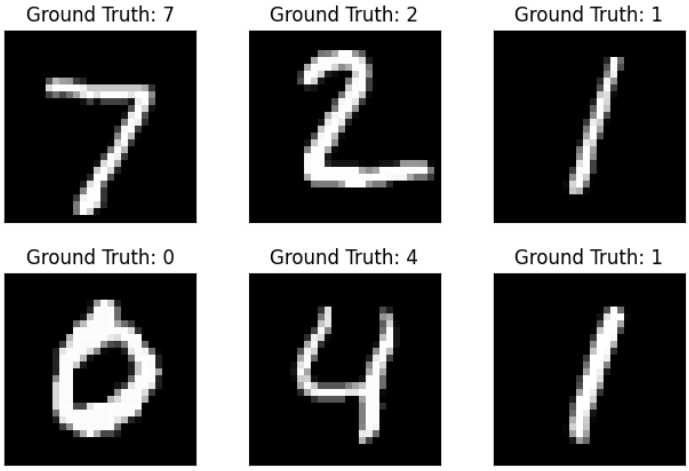

Figure 1: Image of the plot of the first six example digits of the dataset

B. Make your network code repeatable

The network code was made repeatable to ensure consistency and ease of use. This was done by setting torch.manual_seed(1), and by turning off CUDA using torch.backends.cudnn.enabled = False.

C. Build a network model

Similar to the example in the tutorial, a network was created,

- 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.

Figure 2 and Figure 3 show the Class where the network is defined, and the output of print(network). Figure 4 shows the diagram of the network.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.dropout = nn.Dropout2d(p=0.5)
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.dropout(self.conv2(x))))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)
```

Figure 2: Image of the defined network model

```
Net(
  (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)
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (dropout): Dropout2d(p=0.5, inplace=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Figure 3: Image of output of print(network)

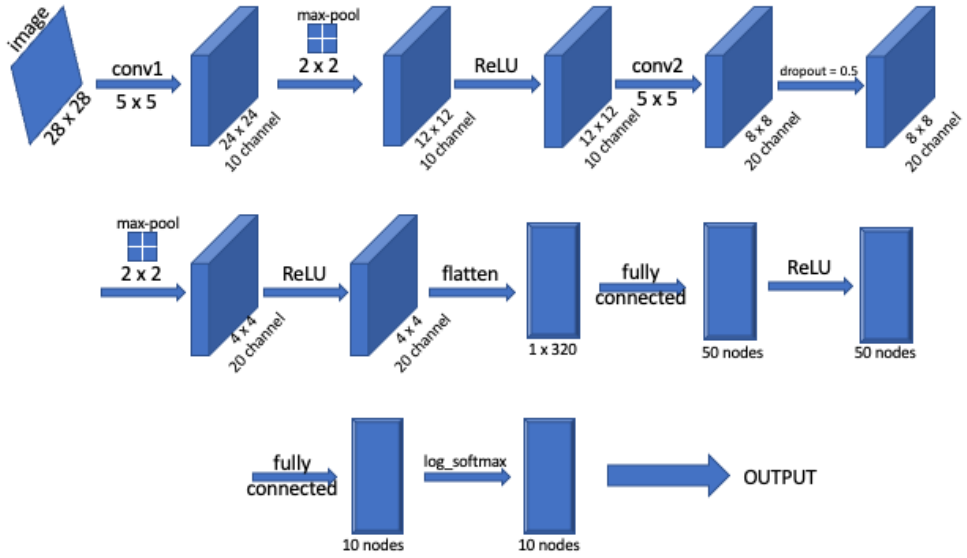

Figure 4: Diagram of the network

D. Train the model

The model is trained for 5 epochs. After each epoch the model is evaluated on both the training and test sets. A batch size of 64 is used. The accuracy scores are then collected and the training and testing accuracy is plot in a graph as seen in Figure 5. As it can be seen, the train and test loss both decrease as the model learns more from the dataset after successive epochs.

After the 5 epochs, by running the trained model on the test set, the accuracy obtained was 9799/10000 or 98%, with an average loss of 0.0604.

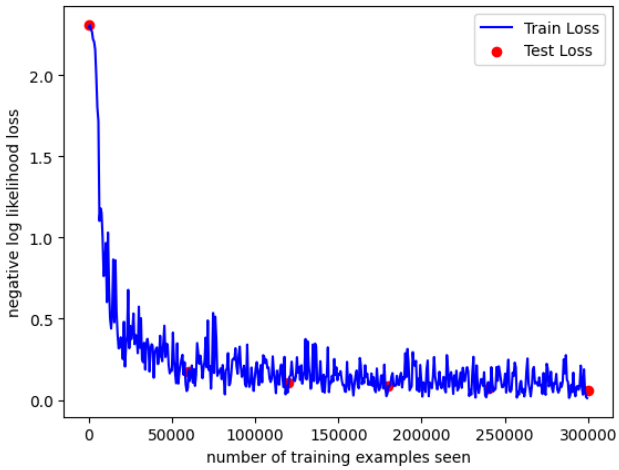

Figure 5: A plot of the training and testing loss vs the number of training images passed through the model

E. Save the Network to a file

After the training is complete, the network with the weights are saved as a "model.pth" file using the torch.save(network.state_dict()) function.
This saved network file is then used for the next tasks.

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

The network is read, and the model is run on the first 10 examples in the test set. The network is first set to evaluation mode. After passing the 10 samples of the test set through the network, we obtain the results as shown in Figure 7. The network classifies all the 10 images correctly.
Figure 6 shows the 10 output values (only 2 decimal places), the index of the max output value, and the correct label of the digit.

```
Example 0: Correct Label: 7
Output[0]: -16.13
Output[1]: -18.53
Output[2]: -9.23
Output[3]: -11.23
Output[4]: -21.73
Output[5]: -19.75
Output[6]: -33.22
Output[7]: -0.00
Output[8]: -15.30
Output[9]: -12.92

Example 2: Correct Label: 1
Output[0]: -14.66
Output[1]: -0.00
Output[2]: -8.84
Output[3]: -11.65
Output[4]: -8.62
Output[5]: -9.92
Output[6]: -11.07
Output[7]: -8.19
Output[8]: -8.50
Output[9]: -11.20

Example 4: Correct Label: 4
Output[0]: -15.04
Output[1]: -21.21
Output[2]: -13.16
Output[3]: -15.31
Output[4]: -0.00
Output[5]: -14.41
Output[6]: -15.26
Output[7]: -13.77
Output[8]: -13.87
Output[9]: -5.99

Example 6: Correct Label: 4
Output[0]: -19.14
Output[1]: -12.15
Output[2]: -15.03
Output[3]: -14.10
Output[4]: -0.02
Output[5]: -9.42
Output[6]: -19.30
Output[7]: -9.56
Output[8]: -4.16
Output[9]: -5.76

Example 8: Correct Label: 5
Output[0]: -14.49
Output[1]: -23.31
Output[2]: -16.03
Output[3]: -15.56
Output[4]: -16.06
Output[5]: -0.00
Output[6]: -8.40
Output[7]: -19.02
Output[8]: -7.06
Output[9]: -8.67

Example 1: Correct Label: 2
Output[0]: -9.47
Output[1]: -8.87
Output[2]: -0.00
Output[3]: -16.51
Output[4]: -19.73
Output[5]: -20.56
Output[6]: -14.07
Output[7]: -23.81
Output[8]: -12.36
Output[9]: -25.26

Example 3: Correct Label: 0
Output[0]: -0.00
Output[1]: -23.02
Output[2]: -15.63
Output[3]: -20.58
Output[4]: -25.15
Output[5]: -15.82
Output[6]: -13.85
Output[7]: -19.17
Output[8]: -19.28
Output[9]: -14.89

Example 5: Correct Label: 1
Output[0]: -18.91
Output[1]: -0.00
Output[2]: -12.65
Output[3]: -15.30
Output[4]: -10.94
Output[5]: -14.75
Output[6]: -17.04
Output[7]: -9.65
Output[8]: -11.03
Output[9]: -13.69

Example 7: Correct Label: 9
Output[0]: -21.93
Output[1]: -16.62
Output[2]: -12.64
Output[3]: -8.78
Output[4]: -7.18
Output[5]: -10.19
Output[6]: -25.77
Output[7]: -12.63
Output[8]: -10.84
Output[9]: -0.00

Example 9: Correct Label: 9
Output[0]: -17.98
Output[1]: -24.72
Output[2]: -18.15
Output[3]: -15.80
Output[4]: -11.71
Output[5]: -14.39
Output[6]: -28.23
Output[7]: -7.21
Output[8]: -8.59
Output[9]: -0.00
```

Figure 6: Screenshot from the output, showing the 10 output values, the index of the max output value and the correct label of the digit

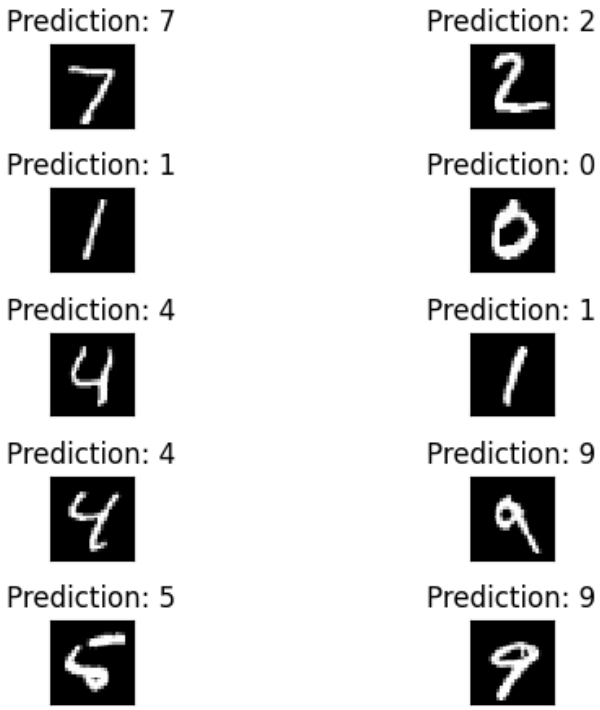

Figure 7: A grid of the first 10 digits being classified correctly by the trained model

G. Test the network on new inputs

The 10 digits [0-9] are written on a piece of paper using a marker. Each digit is cropped to its own square image and is scaled to 28 x 28. The images are then loaded into our code using a dataloader called digit_loader. When these images are being loaded, they are converted from RGB to greyscale, and inverted, to match the MINST images, as the MINST dataset has images of white digits on a black background.

Figure 8 shows the grid of the model correctly classifying all handwritten digits from 0 - 9 correctly.

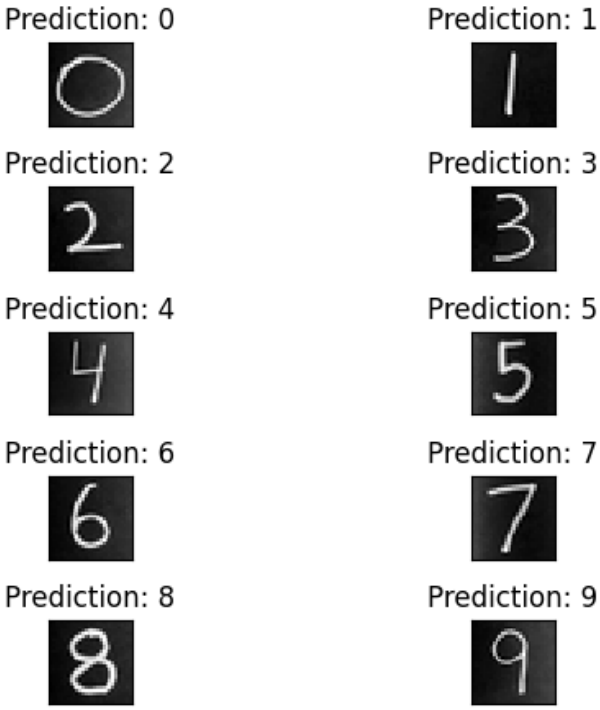

Figure 8: A grid of all handwritten digits being classified correctly by the trained model

Task 2: Examine the network

A. Analyze the first layer

We define a main function that loads a pre-trained model (model.pth) from the MNIST dataset, initializes a new network, loads the test data using the DataLoader class, defines an optimizer, and prints the shape of the first convolutional layer's weights using the optimizer. A 2x5 grid containing the first ten filters' weights and shapes is plotted using pyplot functions. Figure 9 shows the plot of the 10 filters.

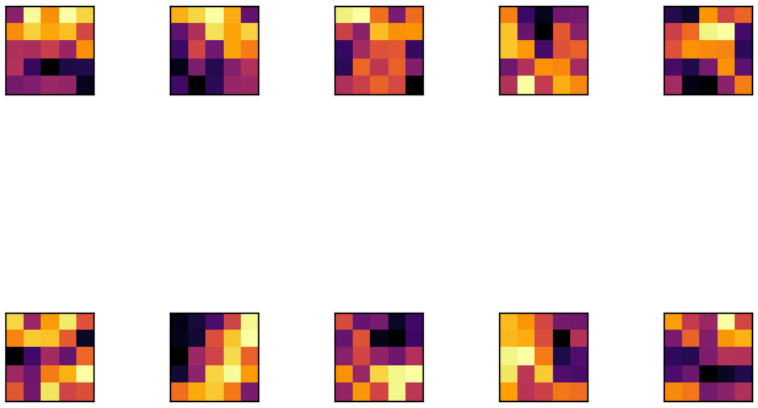

Figure 9: A plot of the 10 filters, 5x5 in size

B. Show the effect of the filters

We load the first image from the test set and we retrieve the weights from the first convolutional layer based on the weights of the first convolutional layer. A plot grid is defined as a grid that has four rows and five columns. In order to apply each filter to the image using the OpenCV filter2D function, a loop iterates over the ten filters and applies them to the image one at a time. In the upper half of the subplot, we plot the filtered image to its corresponding subplot, and in the lower half of the subplot, we plot the weight of the filter against itself. The images obtained from filter weights show that the filters part having dark color pixels imply high contrast in the actual image and the parts with light color have lower contrast in the image. This procedure can be used to classify or enhance certain features of an image and reveal important information from the image. Figure 10 shows the plot of the 10 filters and the filters applied on the first image of the test set.

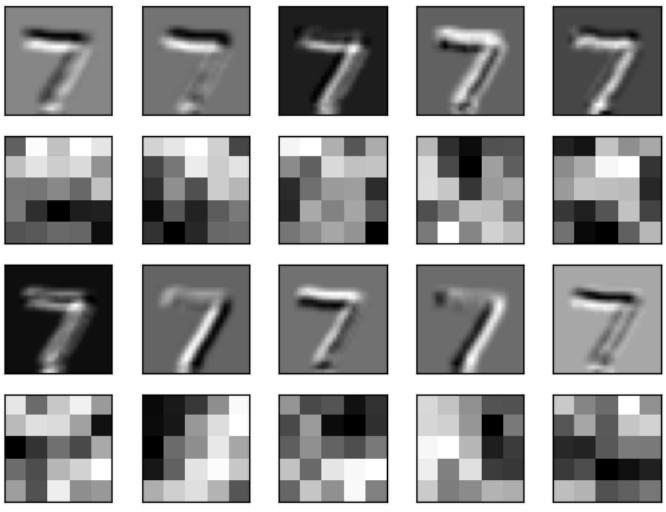

Figure 10: A plot of the 10 filtered images

Task 3: Transfer Learning on Greek Letters

The code has been designed to classify images of Greek letters into the following three categories: Alpha, beta, and gamma.

In the first step of the function, the parameters for training are set, including the number of epochs, the learning rate, the momentum, and the log interval. It also sets a random seed for code reproducibility.

In the next stage, the network and optimizer are defined as instances of the Network class, where the network is defined as an instance of the Network class in the following section.

The code then loads a pre-trained model from a saved file model.pth that contains the parameters of the previously trained MNIST network that has been saved. Printing the network of the model will enable you to view the model's architecture.

Following this, the training images are loaded into the network using ImageFolder, and they are transformed (GreekTransform) by cropping the image and inverting it. Additionally, the tensor values are normalized so that a mean of 0.1307 is obtained and a standard deviation of 0.3081 is obtained. The training dataset is then loaded using DataLoader from torch.utils.data, which returns a batch of images with their corresponding labels as an output. We repeat the same process for the test dataset that we created which contains 4 images for each class (a total of 12 images).

As a next step, the code selects a batch of images from the test dataset and plots them with the help of matplotlib.pyplot. The images are accompanied by ground truth labels that correspond to the images.

Afterward, the network parameters are frozen, and the last layer of the network is replaced with a linear layer with 3 nodes in order to replace the previous layer. The new layer consists of a nn.Linear module with 50 input features and 3 output features. To verify that the changes have been made, the network is printed again.
Using the matplotlib.pyplot package, the train and test losses are then plotted in order to analyze them.
To conclude, we select a batch of images from the test dataset and plot them with their predicted labels that have been obtained by passing the images through the network and selecting the index of the maximum value in the output tensor as the predicted label.

After running the model for 20 epochs, we have been able to perfectly identify all the letters, as seen in Figure 11.

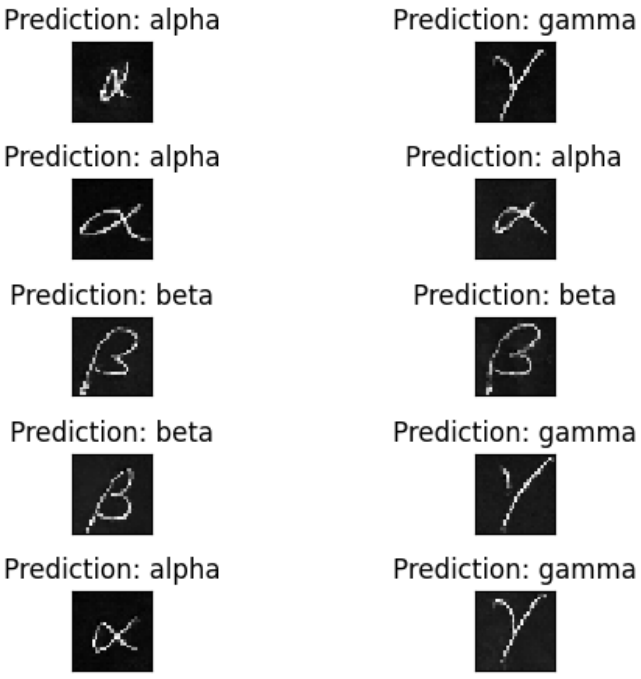

Figure 11: A grid of all handwritten Greek letters being classified correctly by the trained model

Figure 12 shows the plot of training error.

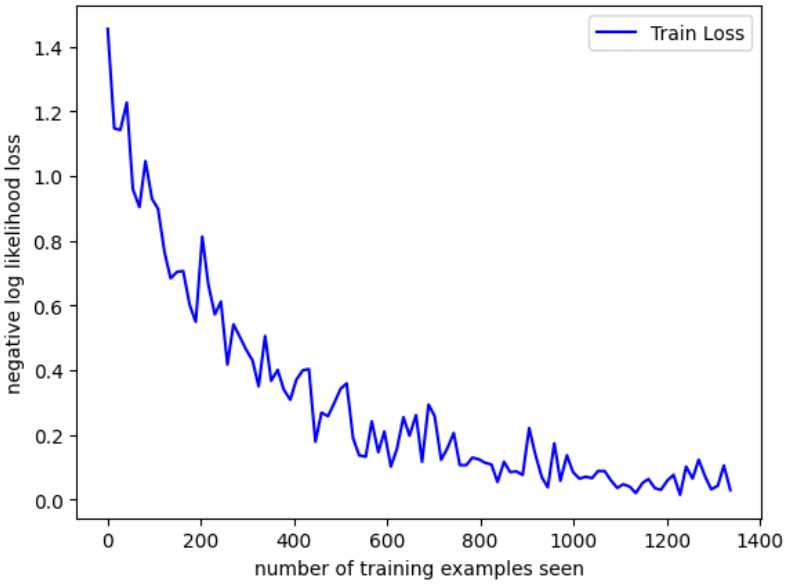

Figure 12: A plot of the Training error for 20 epochs

Figure 13 shows the printout of the modified network

```
Net(
  (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)
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (dropout): Dropout2d(p=0.5, inplace=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=3, bias=True)
)
```

Figure 13: Image of output of print(network) of the modified network

Task 4: Design your own Experiment

The goal of this Task is to undertake some experimentation with the deep Network for the MNIST Task. However, in this task, the Fashion MNIST dataset is used instead of the digit dataset so that we can clearly see the effect of changes in the network.

A. Develop a plan

We plan to change three dimensions of the network.
L = Number of Epochs
M = The dropout rates of the dropout layer
N = Number of neurons in the fully connected linear layers fc1 and fc2

The entire model was trained by varying these 3 parameters L, M, N for a total of 4 * 3 * 5 = 60 variations.
The range of L or Number of Epochs = [3,4,5,6]
The range of M or the dropout rates of the dropout layer = [0.3,0.5,0.7]
The range of N or number of neurons in the fully connected linear layers fc1 and fc2 = [20,40,60,80,100]

B. Predict the Results

Epochs: When the number of epochs is increased, it generally allows the model to train longer and potentially converge to a better solution. However, too many epochs can lead to overfitting, where the model becomes too specialized to the training data and performs poorly on new, unseen data. On the other hand, too few epochs may result in an underfit model that doesn't capture the complexity of the data. Therefore, the optimal number of epochs depends on the specific dataset and model architecture.

Dropout Rates: Adding dropout to a neural network can improve its generalization performance by preventing overfitting. Increasing the dropout rate can make the model more robust to noise and increase the regularization effect. However, if the dropout rate is too high, the model may lose too much information and underfit the data. Therefore, the optimal dropout rate depends on the complexity of the dataset and the size of the model.

Number of neurons in the fully connected linear layers: Increasing the number of neurons in the fully connected layers can increase the capacity of the model and allow it to capture more complex patterns in the data. However, adding too many neurons can also lead to overfitting, especially if the dataset is not large enough. Additionally, increasing the number of neurons can also increase the computation time and memory requirements of the model. Therefore, the optimal number of neurons depends on the size and complexity of the dataset and the model architecture.

C. Execute your plan

After running the loop, we find that the accuracy levels fluctuate a lot over different variations out of the 60 different variations. Most of the variations with low accuracy is due to overfitting of the data.

The most ideal variation with the highest accuracy of 86% obtained was:
L = Number of Epochs = 6
M = The dropout rates of the dropout layer = 0.6
N = Number of neurons in the fully connected linear layers fc1 and fc2 = 60
Figure 14 shows the plot of the train loss and testing loss for the most accurate variation with above values.

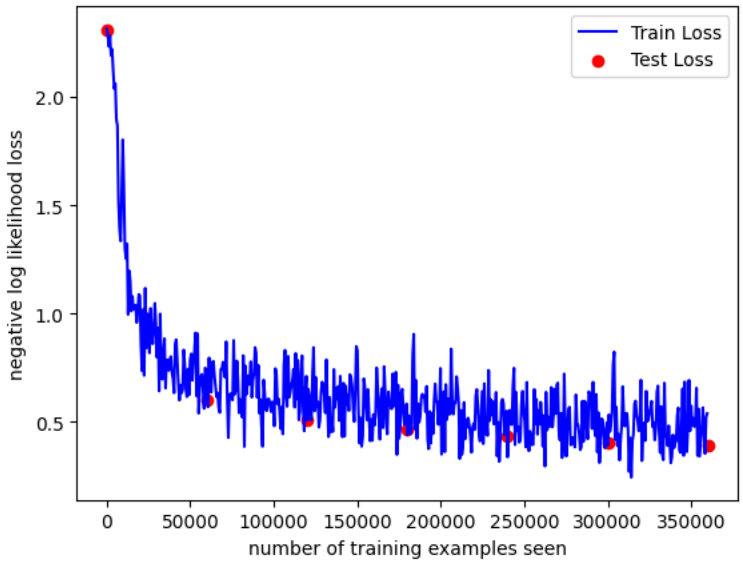

Figure 14: A plot of the training and testing loss vs the number of training images passed through the model

Figure 15 shows the diagram of the optimal variation of the network.

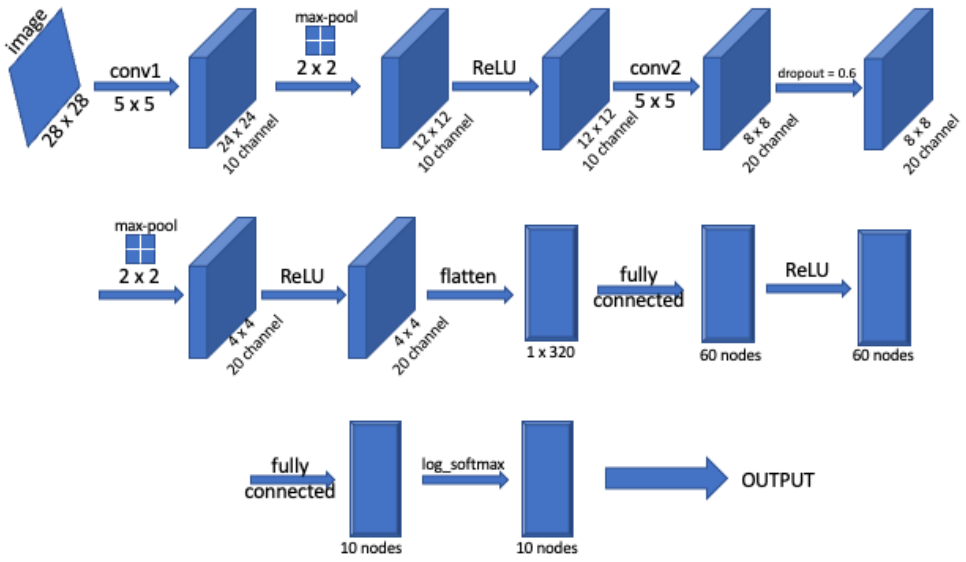

Figure 15: Diagram of the most optimal variation of the network

Extensions

Three extensions have been added to this project. Transfer Learning of Task 3 was applied onto five Greek Letters instead of 3. More dimensions were evaluated for Task 3, and The analysis of the first two convolution layers of a pretrained AlexNet model was analyzed.

Extension 1: Transfer Learning applied on five Greek Letters

The model saved from Task 2 is used for Transfer Learning on three Greek letters. For this extension, the model is trained and tested on images of Greek letters of five classes, ‘alpha’, ‘beta’, ‘gamma’, ‘delta’ and ‘epsilon’. For this extension, the output nodes of the network were changed to 5 instead of 3.
Figure 16 shows the grid of the random test images of the five classes being correctly identified by the model.

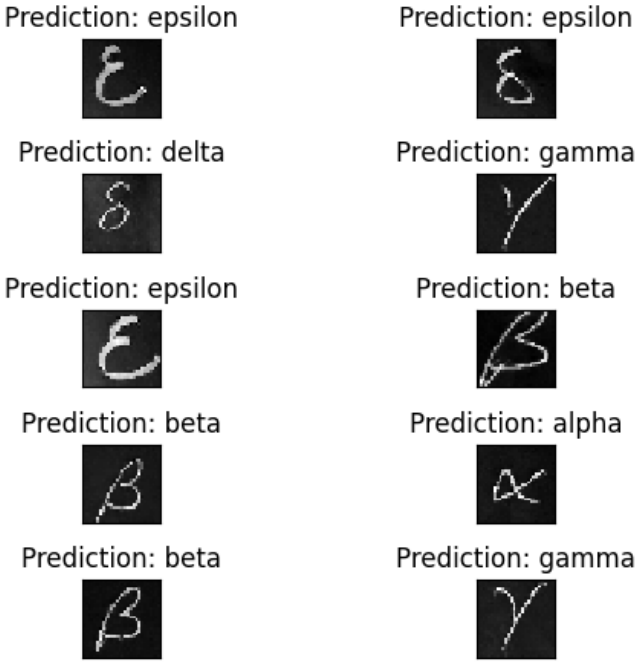

Figure 16: A grid of all handwritten letters across 5 classes being classified correctly by the trained model

Extension 2: Evaluation of more dimensions for Task 3

For the Greek Dataset, two different dimensions were changed, the number of epochs and the batch size. When the number of epochs was increased, the model was trained for a longer period of time, allowing it to potentially learn more complex patterns in the data. However, if the number of epochs was set too high, the model could be overfit to the training data and perform poorly on new data.

When the batch size was increased, the model was trained on more examples at once, which can lead to faster convergence and better generalization. However, larger batch sizes also require more memory and can lead to slower training times or even out-of-memory errors, which was faced when running on a laptop.

In general, what was found in this extension was that adjusting the number of epochs and batch size is a trade-off between model performance and training efficiency.

Figure 17 shows an example of overfitting when the number of epochs was increased beyond a certain number.

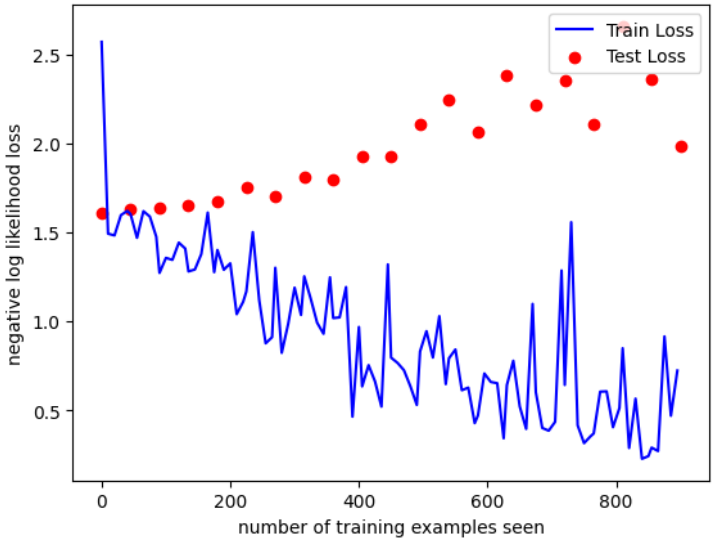

Figure 17: A plot of the training and testing loss vs the number of training images showcasing overfitting

Extension 3: Analysis of pre-trained networks in PyTorch

A pretrained network of AlexNet was taken from the PyTorch package. The CIFAR10 dataset was used to test the model. This network consists of convolution layers in the first layer and the fourth layer. These layers were analysed and the effects of the OpenCV filters are shown. Figure 18 shows the plot of the 10 filters and the 10 filtered images for the first convolution layer. Figure 19 shows the plot of the 10 filters and the 10 filtered images for the second convolution layer.

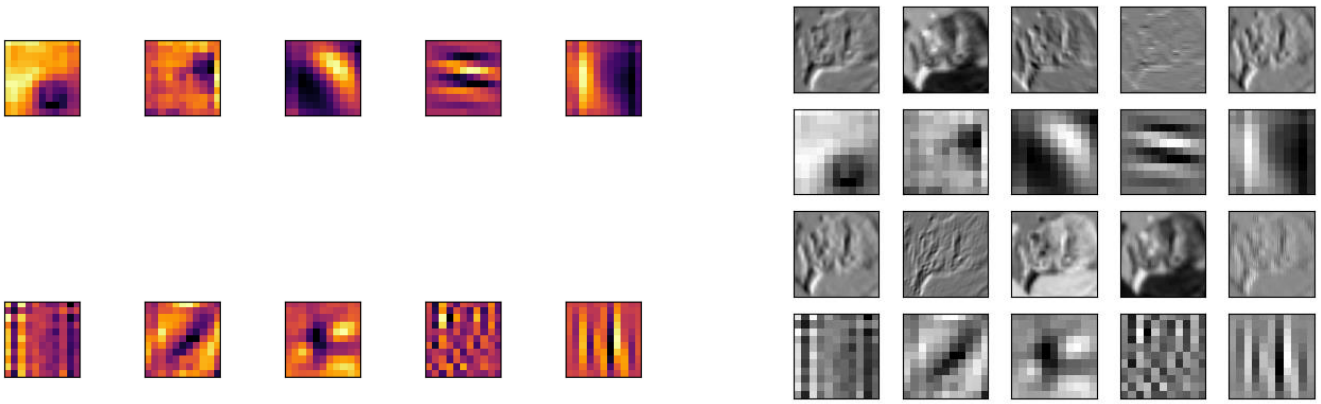

Figure 18: A plot of the 10 filters, and those 10 filters applied on the first image of the CIFAR10 Dataset for the first convolution layer

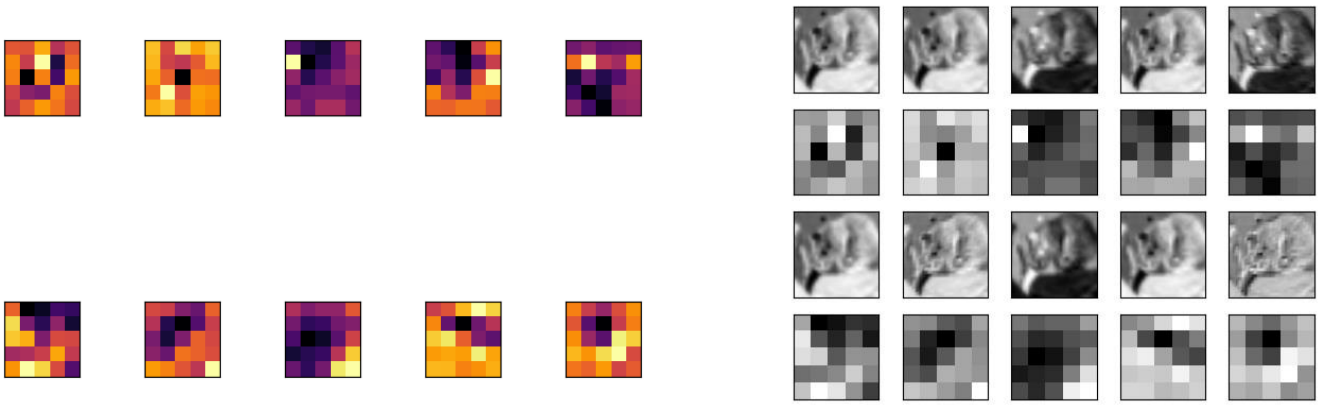

Figure 19: A plot of the 10 filters, and those 10 filters applied on the first image of the CIFAR10 Dataset for the second convolution layer

Project Learnings and Insights

With the help of this project, we got to learn about the deep learning framework "pyTorch", as a stepping stone into the deep leaning world. Among the many pre-trained networks, we have explored the MNIST digit dataset and implemented the basics of deep learning of building a network, training the network on train dataset, testing the network on test dataset, saving the network model to be read and manipulated in the future. Also, the analyzing of various convolutional layers and the model's filter weights have been experimented to understand how they look and how the gradients of the images change. The same model has been used to recognize and predict the Greek letters from the dataset created. Additionally, we had a chance to create small datasets and get to know the procedure of cropping and resizing in order for the model to recognize it better. The visualization of the losses was also something that we learnt under this project., This was a great stepping stone in the deep learning world giving us a kick off to explore more.

Acknowledgements and Resources

We would like to acknowledge Professor Bruce Maxwell and all the Teaching Assistants for their valuable insights and support throughout the project.

- OpenCV Tutorials: https://docs.opencv.org/4.5.1/index.html
- PyTorch Tutorial: https://nextjournal.com/gkoehler/pytorch-mnist
- PyTorch: https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html
- https://towardsdatascience.com/pytorch-conv2d-weights-explained-ff7f68f652eb
- https://www.kaggle.com/code/scratchpad/notebook8a4b19f71b/edit