REPORT
PROJECT 05

(Recognition using Deep Networks)

By – Veditha Gudapati
&
Tejasri Kasturi

Introduction:

The project's goal was to use PyTorch to create and evaluate a deep neural network for digit recognition using the MNIST dataset. First, a convolutional neural network (CNN) architecture was built and trained using 10,000 test samples and 60,000 training samples of handwritten digits from the MNIST dataset. Convolutional, max-pooling, dropout, and fully connected layers made up the network design, which culminated in a softmax output layer for classification.

Following the training phase, the network's performance was evaluated on both training and test datasets, measuring accuracy, and plotting training errors. The trained model was then saved to a file for future use. Subsequently, the saved model was utilized to classify handwritten digits, followed by an examination of the network's internal structures and filters. Furthermore, transfer learning was explored by adapting the trained model to classify Greek letters.

Finally, the last task required us to investigate several factors, such dropout rates, filter widths, and layer counts, that impact network performance. The effects of each dimension were hypothesized, and the network alterations were assessed according to the hypotheses. The results were evaluated to see if they supported the original assumptions.

TASK – 1: Build and Train A Network to Recognize Digits

This task aims to implement digit classification using convolutional neural networks (CNNs) on the MNIST dataset. The program defines a CNN model architecture using PyTorch, including functions for training, evaluating, and visualizing training/testing losses. It preprocesses and loads the MNIST dataset, trains the model, evaluates its performance, and saves the trained model. Next, the program focuses on loading a pre-trained model from a saved file, testing it on the MNIST test dataset, and visualizing predictions. It also plots the first 9 digits from the test set along with their predicted labels. The final program for Task 1 defines a similar CNN model architecture and provides functions for preprocessing an image and classifying a digit using the trained model. It loads a pre-trained model from a saved file, classifies digit images located in a specified directory, and plots them along with their predicted labels.

Task 1 (A): Get the MNIST Digit Data Set

The code utilizes the **torchvision** package to import the MNIST dataset, which consists of 60,000 labeled 28x28 training digits and 10,000 labeled 28x28 test digits. This is facilitated by the **datasets.MNIST** class. Additionally, the **load_data()** function applies the **ToTensor()** transformation, converting the data into **PyTorch** tensors.

To visually inspect the first six example digits from the test set, the **show_samples()** function employs Matplotlib's **pyplot** tool. This function creates a figure with a grid layout of two rows and three columns to display the digits. Each digit image is plotted using **plt.imshow()** in grayscale, with its associated label as the subplot title. The axis ticks from each subplot are removed for a cleaner visualization. This process aids in understanding the dataset's content before training the neural network model, ensuring data integrity, and providing insights into potential challenges.

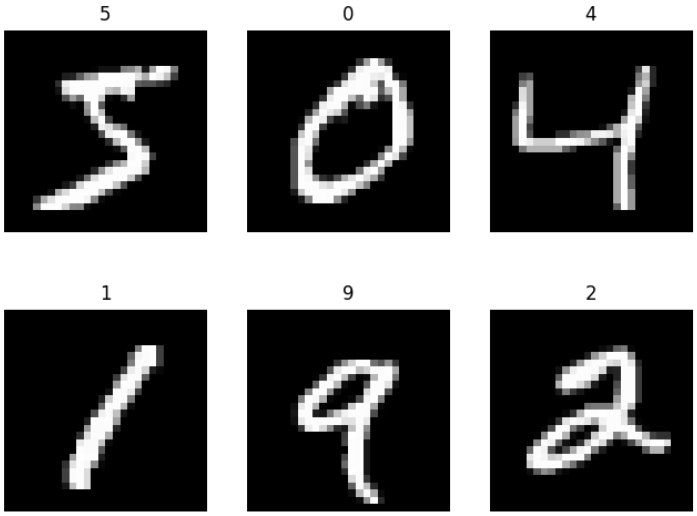

Figure 1: First Six Example Digits

Task 1 (B): Build A Network Model

1. Image (28x28x1): This represents the input layer, where the network receives the image. The image is 28 pixels in width, 28 pixels in height, and has 1 color channel (indicating it's a grayscale image).

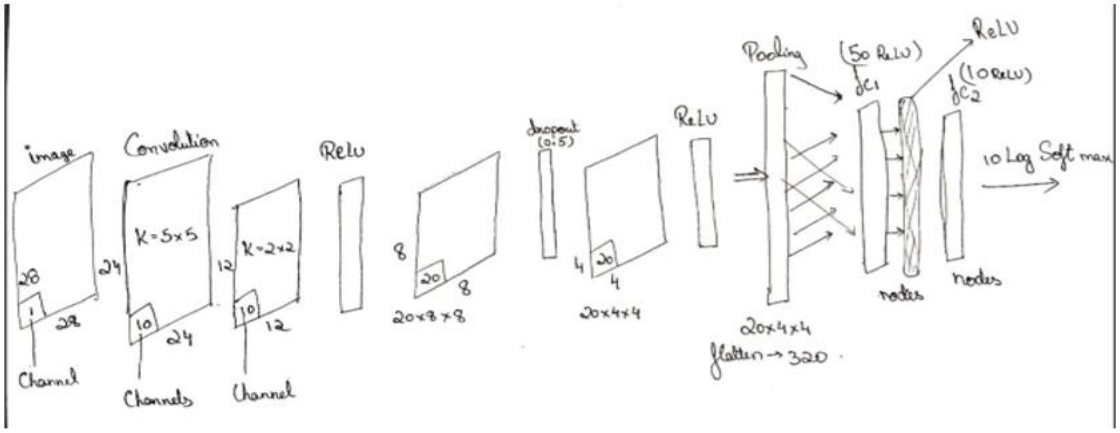

Figure 2:Network Model

2. Conv1 (10x5x5): This is the first convolutional layer. It has 10 filters (or kernels) that are each 5x5 in size. This layer scans the input image with each filter to create a feature map, which helps the network detect features such as edges, corners, etc. The result of this layer is a set of 10 feature maps, and because the filters are 5x5, the spatial dimensions of the output are slightly smaller than the input.

3. ReLU: After pooling, a ReLU (Rectified Linear Unit) activation function is applied. This introduces non-linearity into the network, allowing it to learn more complex patterns. It works by keeping all positive values as they are and converting all negative values to zero.

4. Conv2 (20x4x4): This is the second convolutional layer with 20 filters of size 4x4. Similar to the first convolutional layer, it creates feature maps by convolving the filters with the input. The increased number of filters allows the network to detect a larger variety of features.

5. Dropout (0.5): A dropout layer follows, set at a rate of 0.5. This means that during training, randomly selected neurons are ignored (or "dropped out") at a probability of 50%. This prevents overfitting by forcing the network to not rely too much on any one neuron.

6. Pool (2x2): The second max pooling layer works just like the first one, reducing the dimensions of the feature maps by taking the maximum value in each 2x2 area.

7. Flatten to 320: The flatten operation takes the pooled feature maps and transforms them into a single long vector. This is necessary to connect the convolutional part of the network with the fully connected layers. The number 320 indicates the total number of features in this vector (calculated from the number of feature maps and their dimensions).

8. FC1 (50 ReLU): The first fully connected layer consists of 50 neurons. Each neuron is connected to all 320 features from the flattened vector. The ReLU activation function is again applied to introduce non-linearity.

9. FC2 (10 Log_Softmax): The final fully connected layer has 10 neurons, corresponding to the number of classes the network can classify. The Log_Softmax function is applied to the output of this layer to convert the raw output into log probabilities, which are easier to work with when calculating loss during training.

Task 1 (C): Train the Model

The main function iterates through each epoch using a for loop from 1 to the specified number of epochs (5 in this case). Inside the loop, the **train_network** function is called to train the model for the current epoch. Within **train_network**, the model is trained on the training dataset using the **train_loop** function, which iterates through the batches of training data, updates the model parameters, and computes and stores the training loss at the end of each epoch. Simultaneously, the **test_loop** function evaluates the model's performance on the test dataset, calculating and storing the test loss. Both training and test losses are then plotted against the total number of training examples seen using Matplotlib after all epochs have been completed. Additionally, accuracy scores are computed during each evaluation step within **test_loop** and stored separately for training and testing datasets. While the accuracy scores are not directly used for plotting, they provide valuable insights into the model's performance.

Post-training, the code generates a plot of training and testing errors against the number of training examples seen, aiding in visualizing the learning progress and identifying potential overfitting or underfitting. The blue line depicts the training loss trajectory, while red dots represent test loss values. This visualization enables easy interpretation of how the model's performance evolves over epochs.

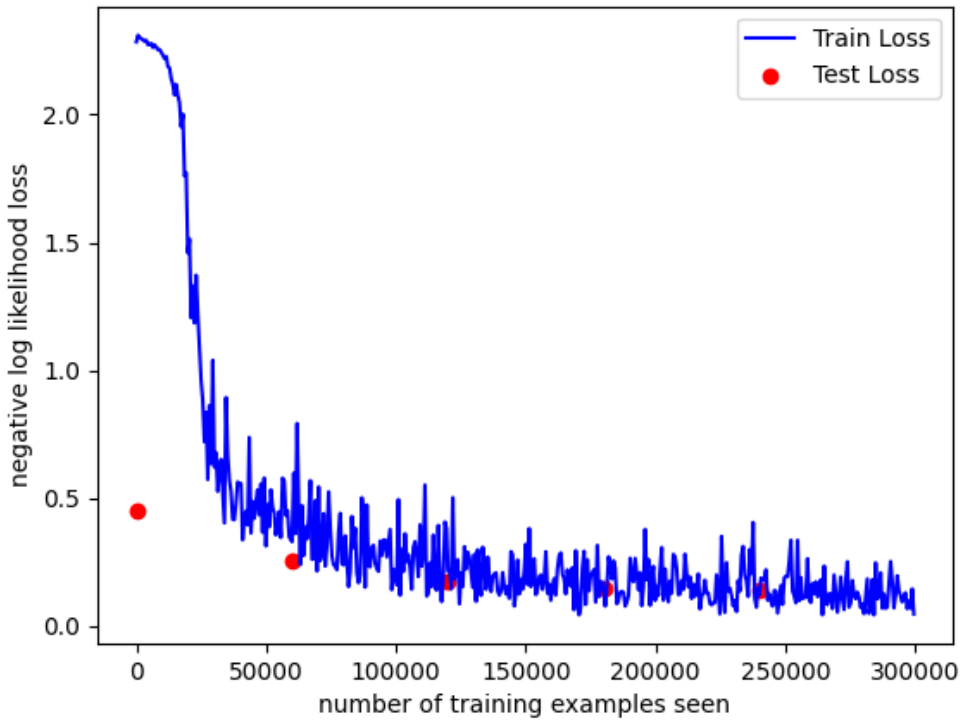

Figure 3:Training and Testing Accuracy

Task 1 (D): Save The Network to A File

Once the training process is completed, the **torch.save()** function is used within the main() function. Specifically**,** **torch.save(model.state_dict()**, **'model.pth'()** is called. This function saves the state dictionary of the model to a file named '**model.pth**'. The state dictionary contains the learned parameters of the model, including weights and biases of each layer.

Task 1 (E): Read the Network and Run It on The Test Set

This task involves testing a neural network model, implemented in Python using **PyTorch**, on the MNIST dataset. Initially, the program imports necessary libraries including **PyTorch**, **torchvision** for dataset handling, and Matplotlib for visualization. The model is loaded from a separate file (task1.py) using **torch.load()**, followed by setting the model to evaluation mode with **model.eval()**. The MNIST test dataset is then loaded using **torchvision's** datasets module, transformed into tensors, and iterated over the first 10 examples. Each example undergoes a forward pass through the model, generating output probabilities for each digit class. The program prints these output values with 2 decimal places, alongside the predicted digit index and the correct label. This process ensures the model's correctness and provides insights into its performance. Additionally, the code plots the first 9 digits along with their predictions, using Matplotlib to visualize the results.

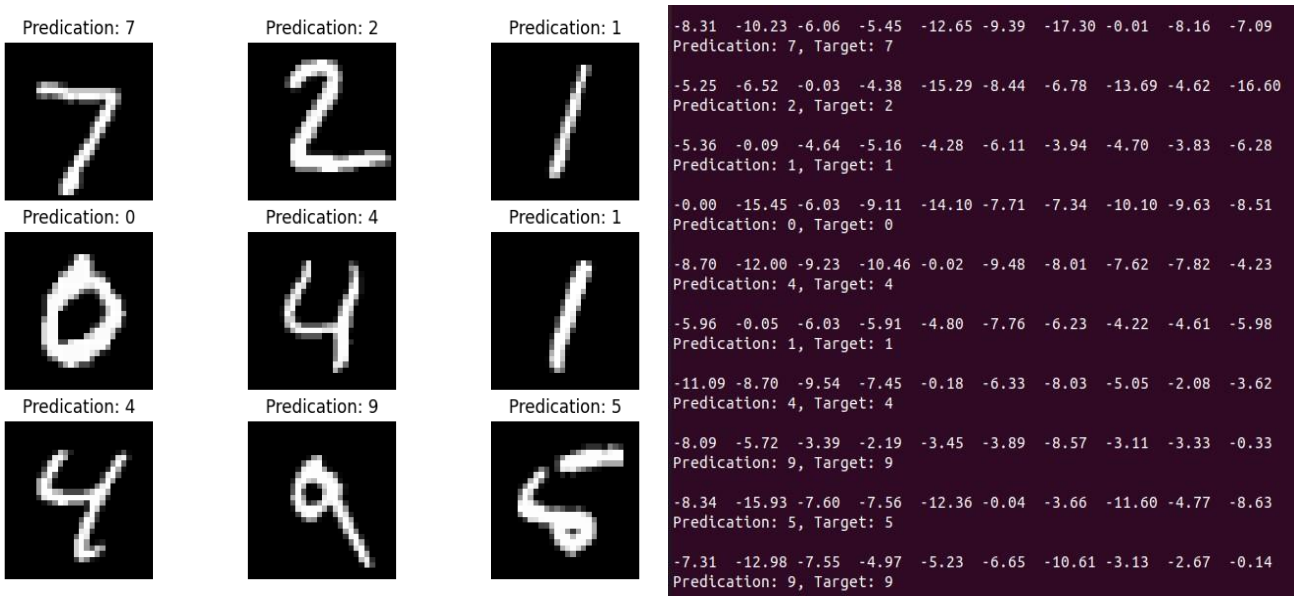

Figure 4: Digits of Test Set & Printed Values

Task 1 (F): Test the Network on New Inputs

In this task we first wrote digits 0-9 on a white surface, ensuring that each digit was clearly distinguishable and clicked pictures of the same. Next, we preprocessed them to match the format expected by the model. Using **ImageMagick**, we resized the cropped images to 28x28 pixels to match the dimensions required by the model.

After resizing, each image was converted to grayscale to simplify the data to a single channel, which is sufficient for recognizing handwritten digits. Additionally, we ensured that the intensity range of the images matched that of the MNIST dataset by adjusting the intensities as necessary.

After pre-processing, we loaded the provided model and fed each preprocessed image into the model for testing. The model generated predictions for each handwritten digit, outputting a probability distribution over the 10 possible classes (digits 0-9). We compared the model's predictions with the actual handwritten digits to assess its performance.

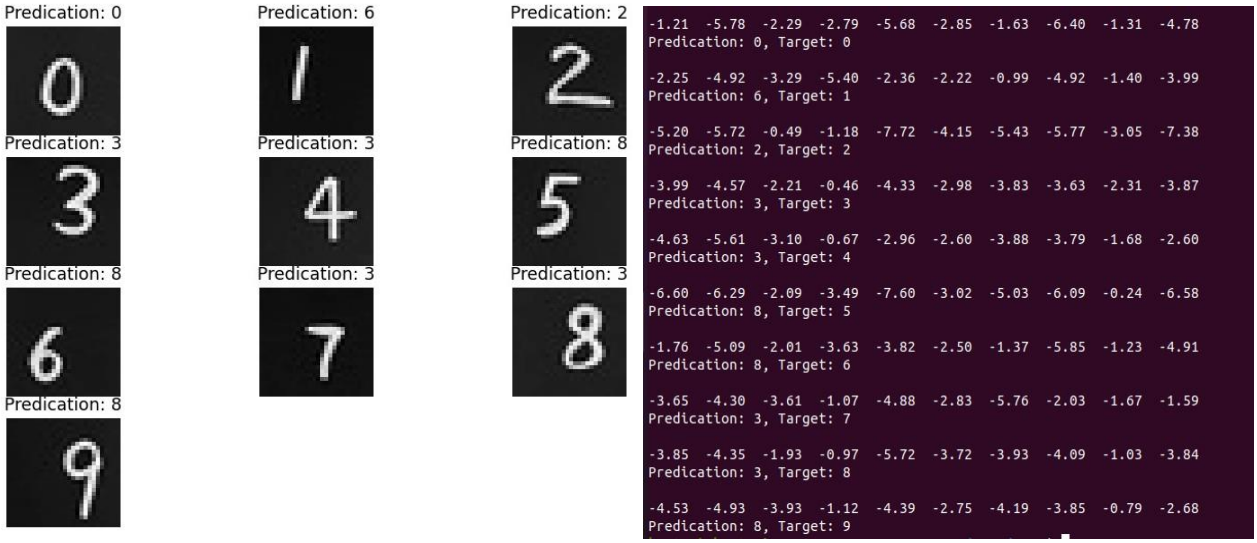

Figure 5: Output of Handwritten Digits and Printed Values

TASK – 2: Examine Your Network

In this task we load a pre-trained neural network model and visualize the weights of its initial convolutional layer. It then applies these weights as filters to an image from the MNIST dataset, using OpenCV's filter2D function, and displays the filtered images alongside their corresponding filters. The script utilizes libraries such as **PyTorch** for deep learning functionalities, **torchvision** for dataset handling, and matplotlib for visualization. By inspecting the model's convolutional layer weights and applying them as filters, it provides insight into how the model processes input images.

TASK – 2 (A): Analyze the First Layer

In this task, accessing the weights of the first layer, named conv1, and visualizing the ten filters using pyplot are implemented as follows:

- **Accessing the weights of conv1:**

  To access the weights of the first convolutional layer, named conv1, the model's architecture and parameters are utilized. Specifically, the weights are accessed using **model.conv1.weight**, where model represents the instance of the loaded neural network model. The conv1 layer is identified by its name as specified during the model definition. Upon access, the weights form a tensor with a specific shape of **[10, 1, 5, 5]**. This shape signifies that there are ten filters in the convolutional layer, each having one input channel and consisting of a **5x5** matrix of weights. The indices **[i, 0]** are employed to access the weights of the **ith** filter within the tensor.

- **Visualizing the ten filters using pyplot:**

  Visualizing the ten filters using pyplot involves creating a grid of subplots, each representing one filter. A figure is first instantiated with a defined size using **plt.figure(figsize=(9,8))**, setting the dimensions of the visualization grid. Within a loop iterating over the range of the number of filters (in this case, 10), each filter is visualized individually using **plt.imshow**. The **imshow** function displays the filter as an image, with the grayscale values representing the weights of the filter. Each subplot represents a single filter, allowing for a detailed examination of its structure and features.

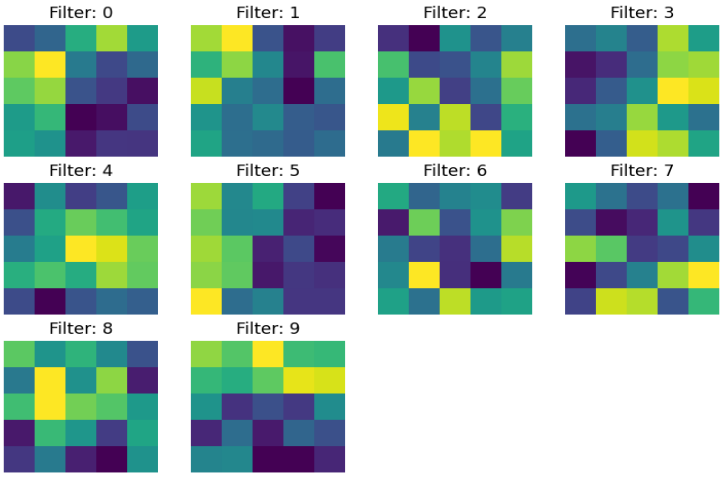

Figure 6:The Output from Visualizing

TASK – 2 (B): Show the Effect of the Filters

In this task, the effect of the ten filters on the first training example image is showcased using OpenCV's filter2D function. To ensure PyTorch doesn't calculate gradients during this process, the **torch.no_grad()** context manager is employed, encapsulating the entire process. Within this context, a loop iterates over the range of ten filters, retrieving each filter from the first layer weights. For each filter, it is converted to a **numpy** array, and then OpenCV's filter2D function is utilized to apply the filter to the input image. This convolution operation results in a filtered output image, which is stored in the variable **filter_img**.

The filtered images, along with their corresponding filters, are visualized using **pyplot**. Within the loop, two subplots are created for each filter: one for visualizing the filter itself and another for displaying the filtered image. By utilizing **plt.imshow** to display the filter and the filtered image, a plot of the ten filtered images is generated.

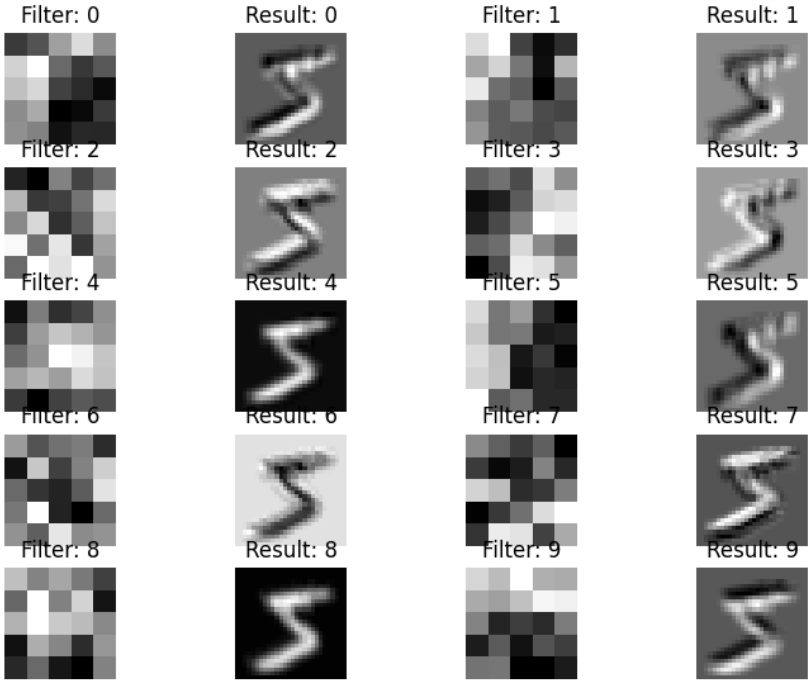

Figure 7: Results with the Effect of the Filters

TASK – 3: Transfer Learning on Greek Letters:

This task implements transfer learning on a Greek letters dataset using PyTorch. It preprocesses data, trains a deep network, and evaluates its performance. Functions handle loading data, training, testing, and visualizing results. Transfer learning enhances model adaptability, crucial for diverse image recognition tasks.

- **GreekTransform Class:**

  The **GreekTransform** class is responsible for defining a series of transformations to preprocess images from the Greek letters dataset. Its **__call__** method takes an input image x and applies a sequence of transformations, including conversion to grayscale, affine transformation, center cropping, and color inversion. By encapsulating these preprocessing steps within a class, it allows for easy application of the transformations to input images using torch vision transforms.

- **loadData Function:**

  The **loadData** function handles loading and transforming the Greek letters dataset using PyTorch's **ImageFolder** class and **DataLoader**. Given the path to the dataset directory (**training_set_path**), it applies the **GreekTransform** preprocessing to the images and creates a **DataLoader** object for efficient data loading and batching during training.

- **train_network Function:**

  The **train_network** function orchestrates the training process of the deep network over multiple epochs. It iterates through the specified number of epochs and calls the **train_loop** function for each epoch. After each epoch, it plots the training loss against the number of training examples seen. This function provides a high-level interface for training the model, allowing users to easily monitor the training progress and visualize the training loss trend over epochs.

- **train_loop Function:**

  The **train_loop** function implements the training loop for a single epoch. It sets the model to training mode, iterates through the data loader, computes predictions, calculates loss, performs backpropagation, and updates model parameters. Additionally, it logs the training loss and computes the accuracy for the current epoch.

- **test_on_new Function:**

  The **test_on_new** function evaluates the trained model on a new Greek letters dataset. It sets the model to evaluation mode, iterates through the test dataset, computes predictions for each image, and plots the image along with the predicted label. This function enables visual inspection of the model's performance on unseen data, providing valuable insights into its generalization ability and effectiveness on real-world tasks.

- **main Function:**

  The main function serves as the main entry point of the script, coordinating the entire workflow. It handles command line arguments to determine the mode of operation, sets random seed for reproducibility, defines dimensions such as learning rate and number of epochs, loads the pre-trained model, initializes the training and testing datasets, defines the loss function and optimizer, initiates the training process using **train_network**, and performs testing using **test_on_new**.

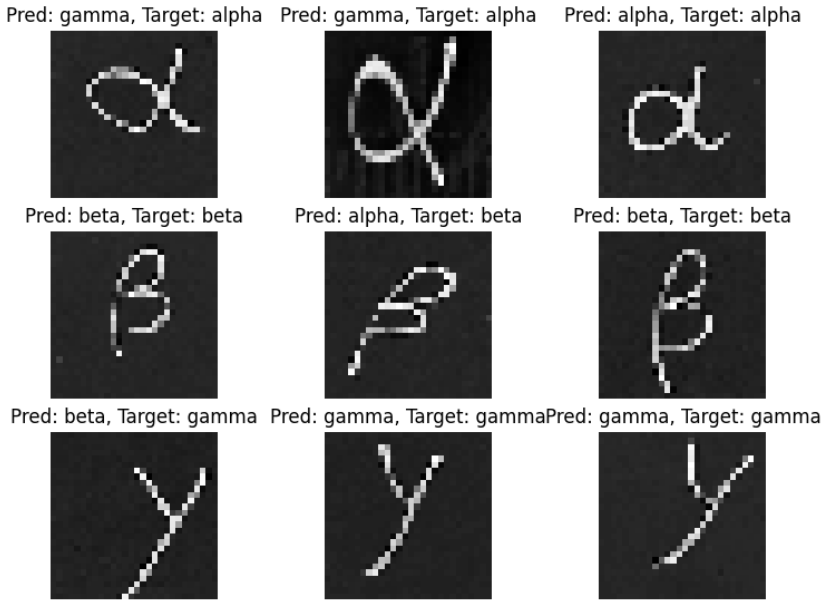

Figure 8: Tested on Provided Dataset

The results for the detection of the Greek letters in the provided dataset is not as expected. The accuracy has increased when additional letters were added to the provided training dataset. The results are shown below :

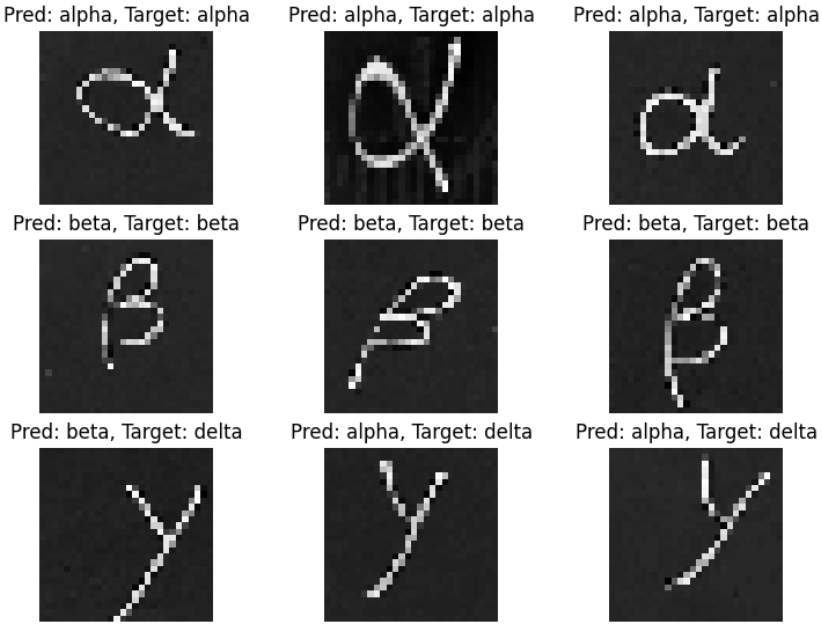

Figure 9: Tested on Addition Alpha, Beta & Gamma

TASK – 4: Compute Features for Each Major Region

TASK – 4 (A): Develop A Plan

Exploration Strategy:

We will use a linear search strategy to explore various combinations of dimensions. For each iteration, we will fix two dimensions and optimize the third, then switch the fixed dimensions in a round-robin fashion. This approach will help us systematically explore the dimension space while avoiding computational overload.

Metrics:

We will primarily focus on the validation accuracy as our main metric to evaluate the performance of each network variation. Additionally, we will also monitor training accuracy, loss, and validation loss to understand the training dynamics and potential overfitting.

Dimensions and Ranges:

The Dimensions and the ranges of the dimensions we would like to explore are as follows:

Number of Convolutional Layers (NCL): 1, 2, and 3 convolutional layers.

Number of Filters per Convolutional Layer (NFL): 16, 32, and 64 filters per layer.

Number of Nodes in the Dense Layer (NDL): 128, 256, and 512 nodes.

Dropout Rate (DR): 0.2, 0.3, and 0.4.

Total Variations:

Total variations = 3 (NCL) * 3 (NFL) * 3 (NDL) * 3 (DR) = 81 variations

Therefore, there are 81 variations in the code.

TASK – 4 (B): Predict the Results

- **Number of Convolutional Layers**: Increasing the number of convolutional layers may lead to better feature extraction and higher accuracy, up to a certain point. However, adding too many layers might cause overfitting.
- **Number of Filters per Convolutional Layer:** Increasing the number of filters can capture more complex patterns in the data, potentially improving accuracy. However, too many filters might increase computational cost without significant gains in performance.
- **Number of Nodes in the Dense Layer**: A higher number of nodes in the dense layer can capture more intricate relationships in the data, potentially leading to improved accuracy. However, increasing the number of nodes excessively may lead to overfitting.
- **Dropout Rate:** Dropout regularization helps prevent overfitting by randomly dropping units during training. A moderate dropout rate may help improve generalization performance, but too high a dropout rate might lead to underfitting.

TASK – 4 (C): Execute your Plan

In this task, he **fashion_mnist** dataset is loaded, and split the dataset into training and testing sets. The **loadData** function is responsible for preprocessing the images. It uses OpenCV (cv2) to resize the images from 28x28 to 32x32 pixels to match the input size expected by the CNN. The resized images are then preprocessed: reshaped to have a single channel (since they are grayscale), normalized to the range [0, 1], and converted to floating-point format.

Next, the data is split into training and validation sets using scikit-learn's **train_test_split()** function. This ensures a portion of the training data is reserved for validation during model training to prevent overfitting. The script defines ranges for dimensions such as the number of convolutional layers, the number of filters per layer, the number of dense nodes in the fully connected layers, and dropout rates. A linear search strategy is employed to explore various combinations of dimensions. For each combination, a CNN model is built using Keras' Sequential API. Training is conducted for a fixed number of epochs (5 epochs in this case) using the training data while monitoring performance on the validation set. After training, the validation accuracy and loss for each epoch are printed to track model performance. The results, including the validation accuracy of each trained model along with its dimensions and training history, are stored in a list called results for later analysis.

Finally, Matplotlib is used to visualize the training and validation accuracy, as well as the training and validation loss, for each model variation. This enables a visual comparison of model performance across different dimension configurations.

Note : For this task, we have added an additional dimension as an extension. Therefore, instead of 3 dimensions the program implements 4 dimensions, they are:

- **num_conv_layers_range:** Number of convolutional layers in the range [1, 2, 3]
- **num_filters_range:** Number of filters in convolutional layers in the range [16, 32, 64]
- **dense_nodes_range:** Number of nodes in dense layers in the range [128, 256, 512]
- **dropout_rate_range:** Dropout rate in the range [0.2, 0.3, 0.4, 0.5]

```
Training model with: Num Conv Layers: 1, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.2
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8790
Epoch 2: Validation Accuracy: 0.8888
Epoch 3: Validation Accuracy: 0.9022
Epoch 4: Validation Accuracy: 0.9019
Epoch 5: Validation Accuracy: 0.9093
Training model with: Num Conv Layers: 1, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.3
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8767
Epoch 2: Validation Accuracy: 0.8888
Epoch 3: Validation Accuracy: 0.8997
Epoch 4: Validation Accuracy: 0.9049
Epoch 5: Validation Accuracy: 0.9058
Training model with: Num Conv Layers: 1, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.4
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8737
Epoch 2: Validation Accuracy: 0.8896
Epoch 3: Validation Accuracy: 0.8987
Epoch 4: Validation Accuracy: 0.9067
Epoch 5: Validation Accuracy: 0.9035
Training model with: Num Conv Layers: 1, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.5
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8740
Epoch 2: Validation Accuracy: 0.8934
Epoch 3: Validation Accuracy: 0.8972
Epoch 4: Validation Accuracy: 0.9000
Epoch 5: Validation Accuracy: 0.9041
```

Figure 10: Results of Convolution Layer 1

```
Training model with: Num Conv Layers: 2, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.2
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8360
Epoch 2: Validation Accuracy: 0.8608
Epoch 3: Validation Accuracy: 0.8702
Epoch 4: Validation Accuracy: 0.8783
Epoch 5: Validation Accuracy: 0.8881
Training model with: Num Conv Layers: 2, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.3
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8422
Epoch 2: Validation Accuracy: 0.8632
Epoch 3: Validation Accuracy: 0.8754
Epoch 4: Validation Accuracy: 0.8767
Epoch 5: Validation Accuracy: 0.8873
Training model with: Num Conv Layers: 2, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.4
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8384
Epoch 2: Validation Accuracy: 0.8592
Epoch 3: Validation Accuracy: 0.8702
Epoch 4: Validation Accuracy: 0.8761
Epoch 5: Validation Accuracy: 0.8822
Training model with: Num Conv Layers: 2, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.5
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8275
Epoch 2: Validation Accuracy: 0.8593
Epoch 3: Validation Accuracy: 0.8694
Epoch 4: Validation Accuracy: 0.8764
Epoch 5: Validation Accuracy: 0.8794
```

Figure 11: Results for Convolution Layer 2

```
Training model with: Num Conv Layers: 3, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.2
Results for each epoch:
Epoch 1: Validation Accuracy: 0.7703
Epoch 2: Validation Accuracy: 0.7989
Epoch 3: Validation Accuracy: 0.8080
Epoch 4: Validation Accuracy: 0.8280
Epoch 5: Validation Accuracy: 0.8413
Training model with: Num Conv Layers: 3, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.3
Results for each epoch:
Epoch 1: Validation Accuracy: 0.7763
Epoch 2: Validation Accuracy: 0.8025
Epoch 3: Validation Accuracy: 0.8157
Epoch 4: Validation Accuracy: 0.8242
Epoch 5: Validation Accuracy: 0.8355
Training model with: Num Conv Layers: 3, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.4
Results for each epoch:
Epoch 1: Validation Accuracy: 0.7790
Epoch 2: Validation Accuracy: 0.8032
Epoch 3: Validation Accuracy: 0.8045
Epoch 4: Validation Accuracy: 0.8282
Epoch 5: Validation Accuracy: 0.8340
Training model with: Num Conv Layers: 3, Num Filters: 16, Dense Nodes: 128, Dropout Rate: 0.5
Results for each epoch:
Epoch 1: Validation Accuracy: 0.7669
Epoch 2: Validation Accuracy: 0.8013
Epoch 3: Validation Accuracy: 0.8130
Epoch 4: Validation Accuracy: 0.8323
Epoch 5: Validation Accuracy: 0.8288
```

Figure 12: Results for Convolution Layer 3

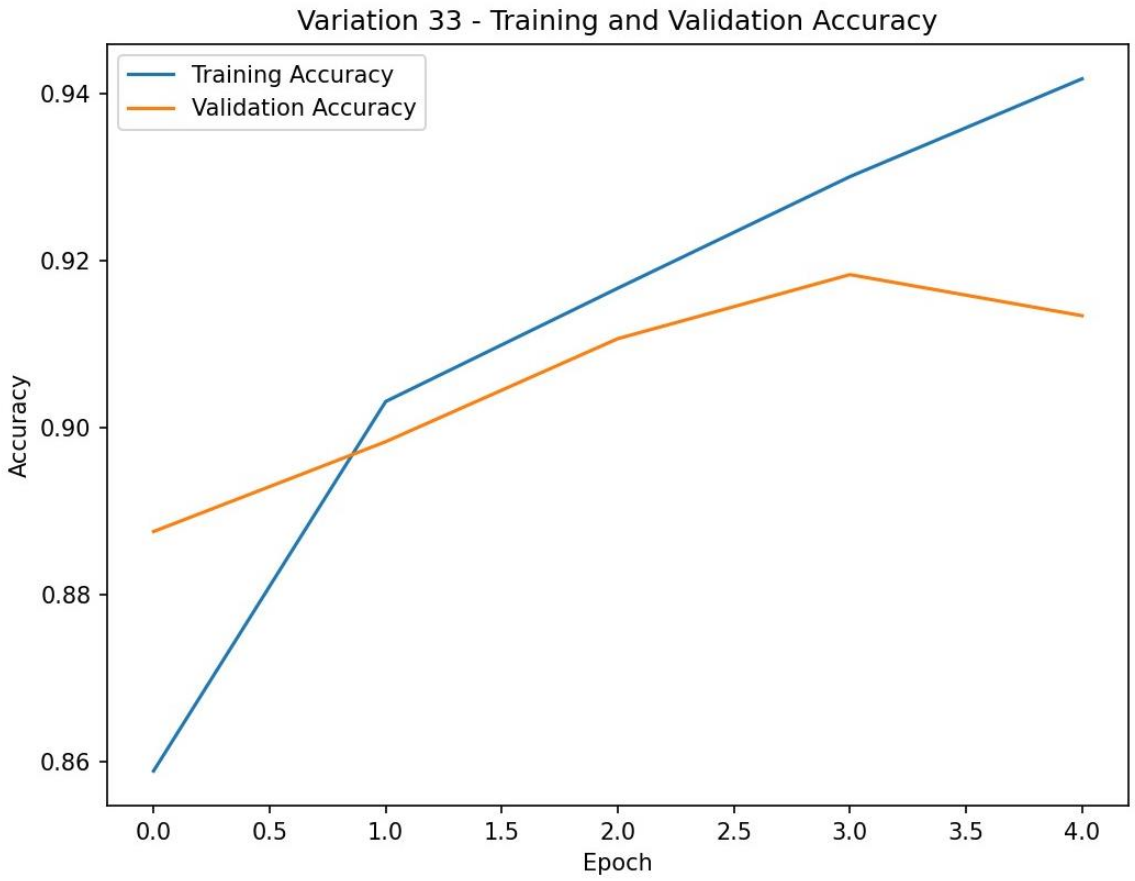

Figure 13: The Plot for the Variation with Highest Validation Accuracy

EXTENSIONS:

1. Live Video Digit Recognition Using Trained Network

The program begins by importing necessary libraries including Torch, OpenCV (cv2), and **torchvision.transforms** for image transformations. It loads a custom neural network model **(MyNetwork)** from a separate Python file **task1.py** and then loads a pre-trained model from the file **model.pth**. Transformation functions are defined to preprocess incoming frames from the camera to match the format of the training data, including grayscale conversion, resizing to 28x28 pixels, and normalization analogous to the MNIST dataset. To apply these modifications to every frame the camera records, the **preprocess_frame** function is developed. The program then enters a loop where it continuously captures frames from the camera, preprocesses them using the defined function, and passes them through the pre-trained model for digit prediction. The predicted digit is overlaid onto the frame using OpenCV's **cv2.putText** function. The loop continues until the user presses **'q'** to quit, at which point the video capture object is released and all OpenCV windows are closed. This implementation creates a real-time digit recognition system using computer vision techniques and deep learning.

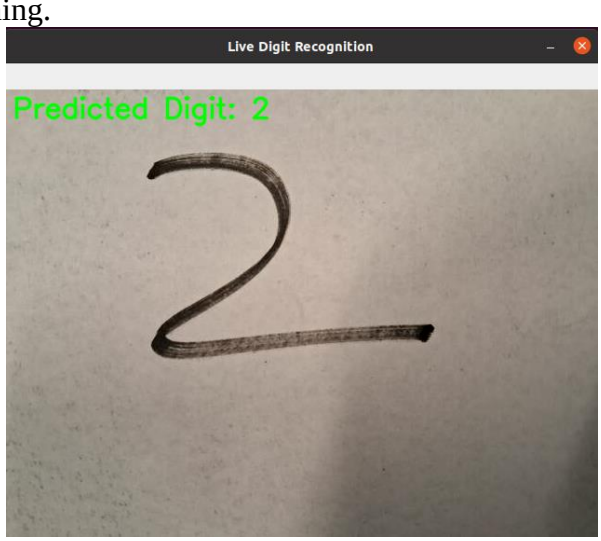

Figure 14: Digit 2 Recognized on Live Video

Figure 15: Digit 7 Recognized on Live Video

Note : The link to the video of the live digit recognition is added to the readme file

2. Visualizing Convolutional Filters and Applying Filter to Image using ResNet-18 and OpenCV

The program implements a visualization and application of filters learned by the first convolutional layer of a pre-trained ResNet-18 model using PyTorch and OpenCV. Initially, the pre-trained ResNet-18 model is loaded, and the weights of its first convolutional layer are extracted. These weights represent the learned filters. The program then visualizes these filters as subplots using matplotlib. Next, an image from the CIFAR10 dataset is loaded and preprocessed, including resizing and conversion to a NumPy array. One of the learned filters is chosen, and OpenCV's filter2D function is utilized to apply this filter to the image. The original and filtered images are then displayed using matplotlib. However, there are potential issues with this implementation, such as the difference in dimensions between CIFAR10 images and the expected input size of ResNet-18, and the color space conversion which might lead to color distortion. To mitigate these issues, additional preprocessing steps such as resizing the images to match the model's input size and handling color space conversion appropriately would be necessary.

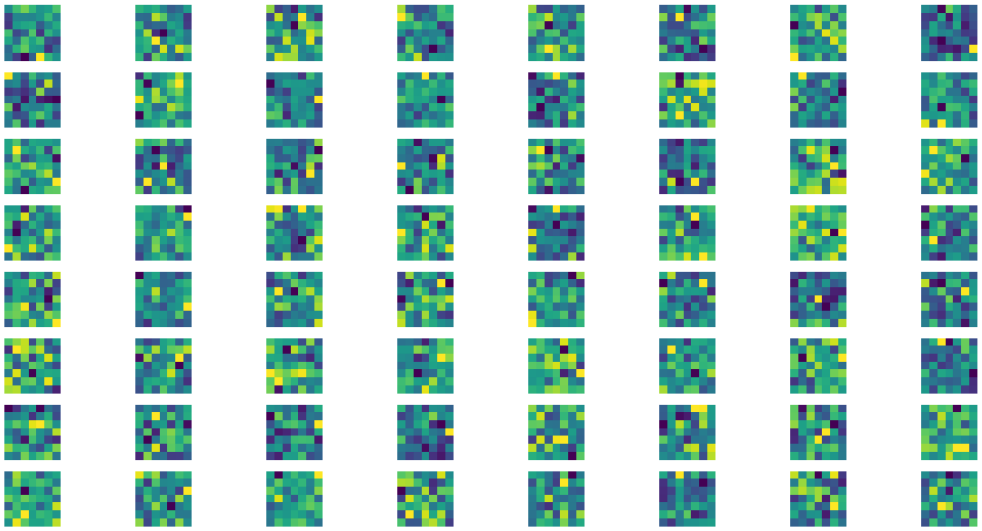

Figure 16:Analyzing First Layer

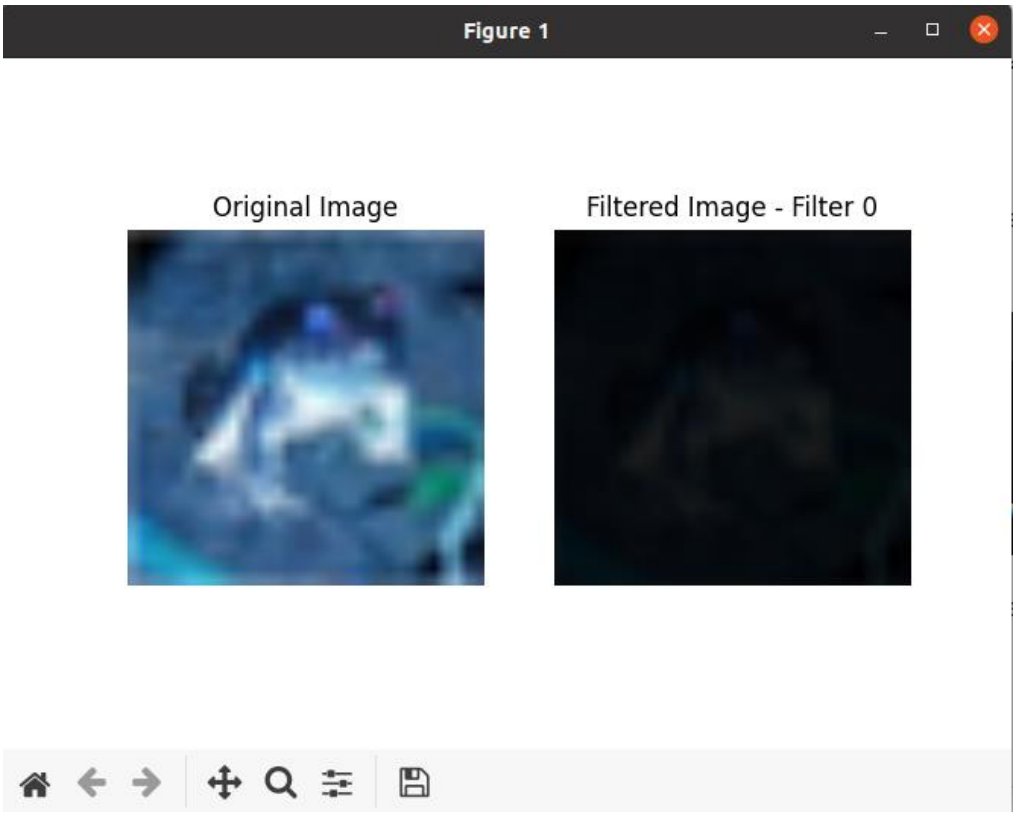

Figure 17:Effects of Filters

3. Task 3 : Detect Additional Letters

As an extension to task 3 , we have added additional Greek letter for detection. The Greek letters that were added for detection are Phi, Delta, and Lambda. The results are as follows:

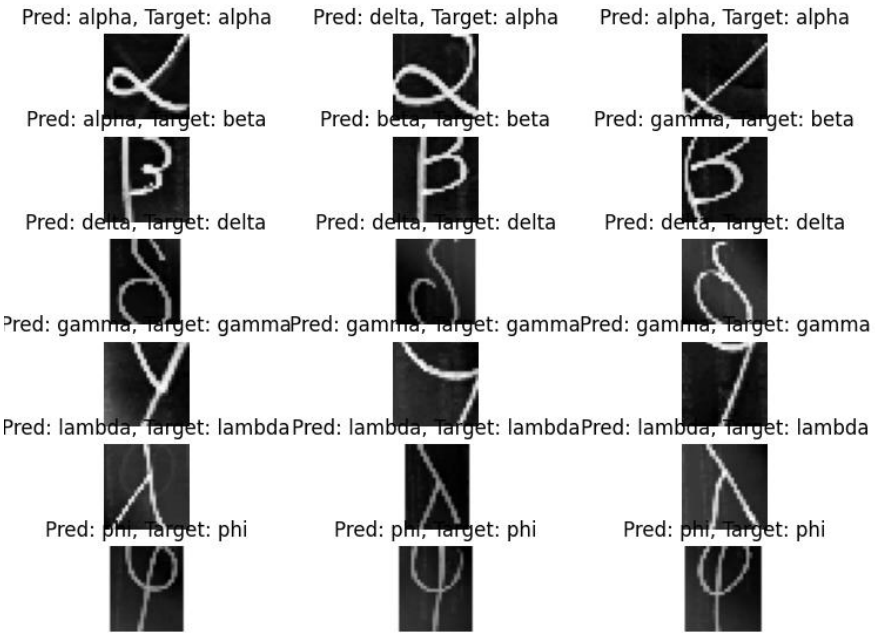

Figure 18:For 10 Epochs - Accuracy 67%

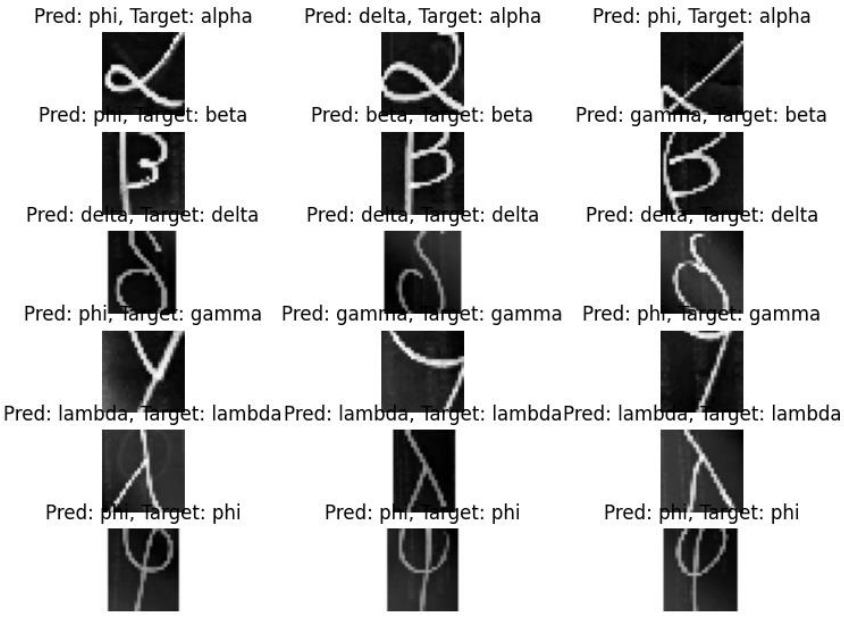

Figure 19: For 20 Epochs - Accuracy 87%

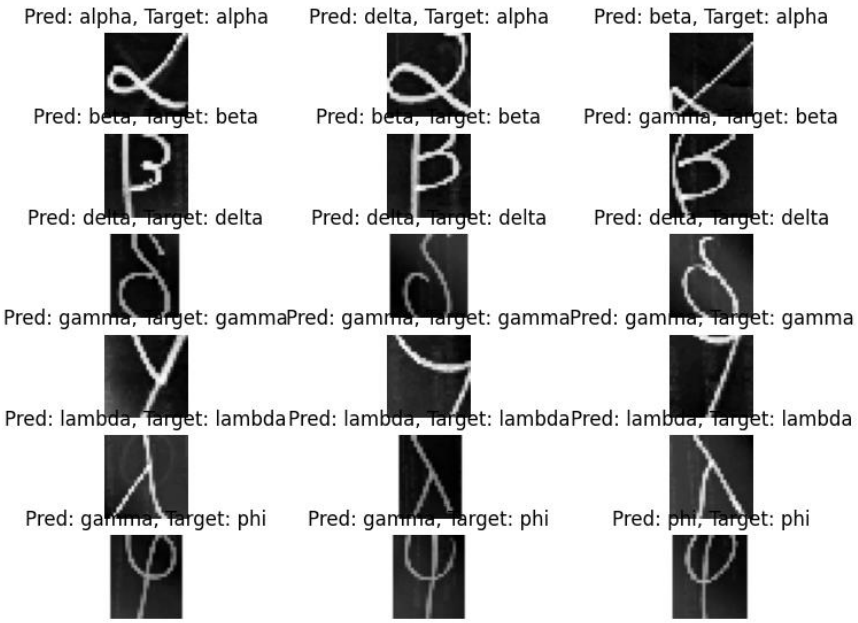

Figure 20:For 25 Epochs - Accuracy 90%

We have observed that as the epochs increase there are variations in the prediction accuracy. The model can correctly recognize.

- Gamma, Delta, Phi and Lambda have 100% accuracy at 10 epochs. While Aplha and Beta only have 67% (2/3) accuracy.
- Phi, Lambda and Delta have 100% accuracy at 20 epochs. While the others have 87% accuracy.
- All the Greek letters except Phi show 100% accuracy in detection, while phi does not show as much accuracy.

4. Task 4 : Add Additional Dimensions

As an extension to task 4 , we have added an additional dimension " Number of Filters" . Therefore, the program implements a total of 4 dimensions instead of 3.

```
Training model with: Num Conv Layers: 3, Num Filters: 64, Dense Nodes: 128, Dropout Rate: 0.3
Results for each epoch:
Epoch 1: Validation Accuracy: 0.8213
Epoch 2: Validation Accuracy: 0.8512
Epoch 3: Validation Accuracy: 0.8658
Epoch 4: Validation Accuracy: 0.8703
Epoch 5: Validation Accuracy: 0.8790
```

Figure 21: Results with 4 Dimensions

REFLECTION ON WHAT WE LEARNT:

1. Model Training and Evaluation: Through training CNN models on the MNIST dataset and evaluating their performance, we learned about key concepts such as loss functions, optimization algorithms, and metrics like accuracy. We also gained practical experience in monitoring training progress, identifying overfitting or underfitting, and optimizing model hyperparameters.

2. Transfer Learning: Exploring transfer learning techniques on the Greek letters dataset expanded our knowledge of model adaptability and reusability. By leveraging pre-trained models and fine-tuning them on new tasks, we understood how to apply deep learning techniques to diverse image recognition tasks effectively.

3. Dimensionality Exploration: The task of exploring different network configurations and optimizing dimensions provided valuable insights into the impact of architecture choices on model performance. By systematically varying parameters such as the number of layers, filters, nodes, and dropout rates, we gained a deeper understanding of how these factors influence model behavior and accuracy.

ACKNOWLEDGEMENTS:

I am grateful to the OpenCV documentation, which has been a vital resource during this project. Understanding and putting different computer vision algorithms and techniques into practice has been made possible by the comprehensive explanations given in the OpenCV documentation. Exploring and implementing tensor flow and keras has been possible by the detailed information provided on their respective websites. The websites referred are: https://pytorch.org/tutorials/beginner/basics/intro.html and https://keras.io/api/.

I would also like to thank my peers for their help and feedback during this project. Throughout this process, exchanging knowledge, working together to troubleshoot challenges, and sharing ideas have all been extremely benefic