Project-5 Report

Name: Priyanshu Ranka (NUID: 002305396)

Semester: Spring 2025

Professor: Prof. Bruce Maxwell

Subject: Pattern Recognition and Computer Vision

1. Project Overview:

This project provides a comprehensive exploration of deep learning methodologies, focusing on the development, training, and analysis of convolutional neural networks for image recognition. Utilizing the MNIST dataset, participants will construct a robust digit recognition system, emphasizing best practices in PyTorch implementation. The project extends beyond basic model creation, encompassing network examination, transfer learning, and experimental design. Participants will also formulate a detailed proposal for a final project, demonstrating their ability to apply learned concepts to novel applications. This project aims to cultivate a deep understanding of deep learning principles, enabling participants to design and evaluate complex neural network architectures effectively.

2. Tasks:

2.1 Build and train a network to recognize digits

Objective: Develop a robust convolutional neural network using PyTorch for MNIST digit recognition. Define the network's architecture, specifying layers, activation functions, and regularization techniques. Implement efficient training and evaluation functions, carefully selecting hyperparameters like learning rate and batch size. Iterate through the dataset, optimizing weights to minimize loss. Ensure modular and well-documented code for readability and maintainability. Save the trained model persistently for subsequent tasks. This phase establishes a foundational model for digit recognition, emphasizing accuracy and efficiency in preparation for advanced explorations.

Approach: Task 1: The approach involves several key steps-

1. Network Architecture Definition:
   - A custom neural network class Net is defined, inheriting from nn.Module.
   - The network comprises two 2D convolutional layers (nn.Conv2d), each followed by ReLU activation and max-pooling (F.max_pool2d).
   - Dropout layers (nn.Dropout2d) are included for regularization.
   - Two fully connected linear layers (nn.Linear) are used for final classification, with ReLU activation on the first and log-softmax on the output.
   - The structure follows a typical CNN pattern, designed to extract features hierarchically from the input images.
2. Data Loading and Preprocessing:
   - The MNIST dataset is loaded using torchvision.datasets.MNIST, with train_loader and test_loader created using torch.utils.data.DataLoader.

- Data transformations, including conversion to tensors and normalization using MNIST's mean and standard deviation, are applied.
- The train loader is set to shuffle the data, and the test loader isn't.

3. Training Procedure:

- The train() function implements the training loop.
- An optim.SGD optimizer is used to update network weights.
- The negative log-likelihood loss (F.nll_loss) is used as the loss function.
- The network's gradients are zeroed, the loss is computed, and backpropagation is performed.
- Training loss and example counts are tracked for plotting.
- The model and optimizer state are saved after each batch of training.

4. Evaluation Procedure:

- The test() function evaluates the trained model on the test dataset.
- The network is set to evaluation mode (network.eval()) to disable dropout.
- The test loss and accuracy are computed.
- The test loss is stored for plotting.

5. Main Execution and Plotting:

- The main() function orchestrates the training and evaluation process.
- The network and optimizer are instantiated.
- The training and testing loops are executed for a specified number of epochs.
- Training and testing losses are plotted to visualize the training process.
- The plot shows the training and testing losses over the training period.
- The trained model is saved to a file called model.pth.
- The code is structured with functions and a main() function, adhering to good Python practices.
- The code is written to be able to be run from the command line.

Plots / Results:

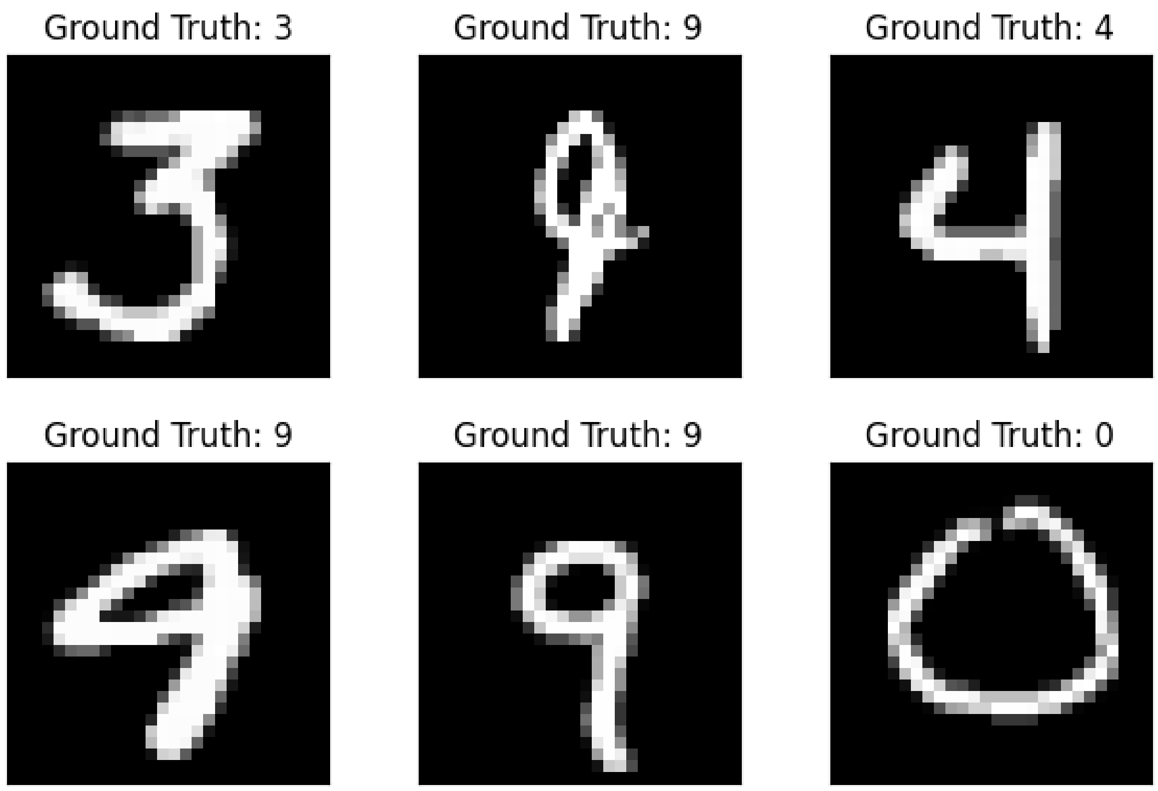


Figure 1A: A plot of the first six example digits from the test set

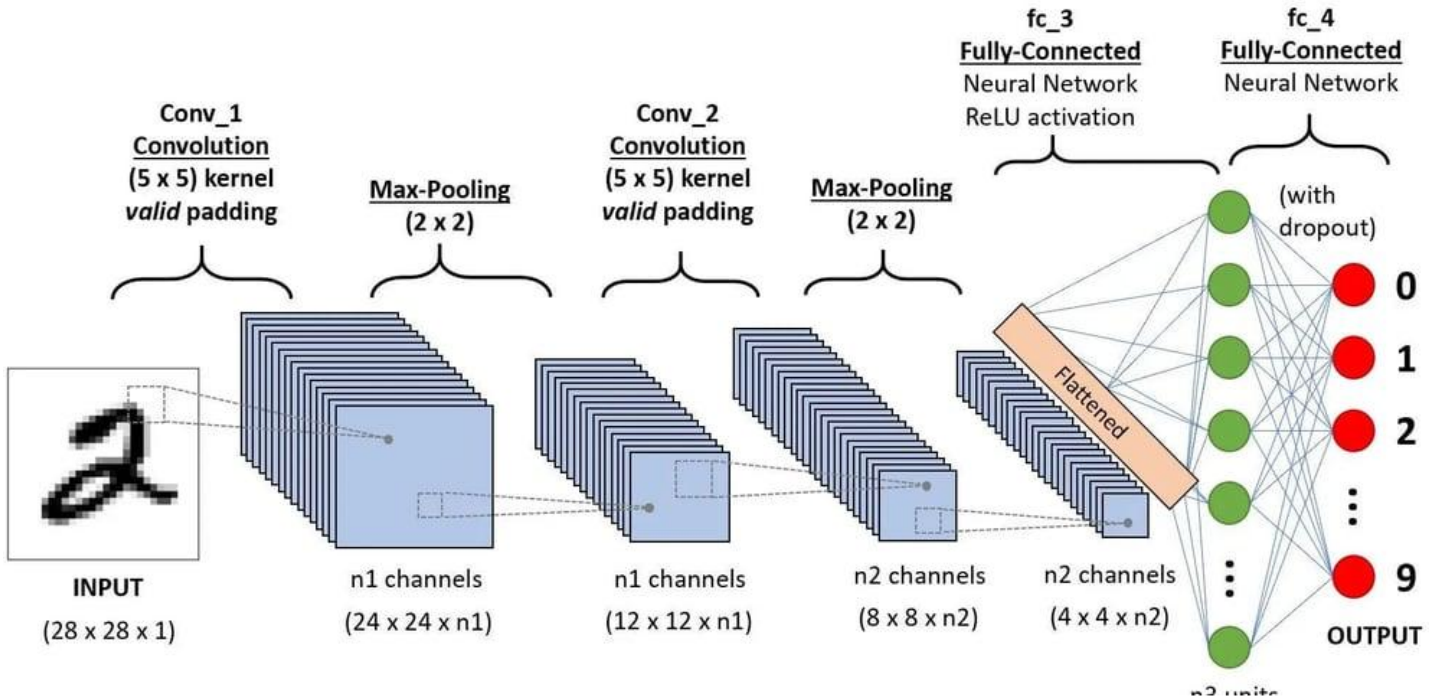


Figure 1B: A diagram of the Network

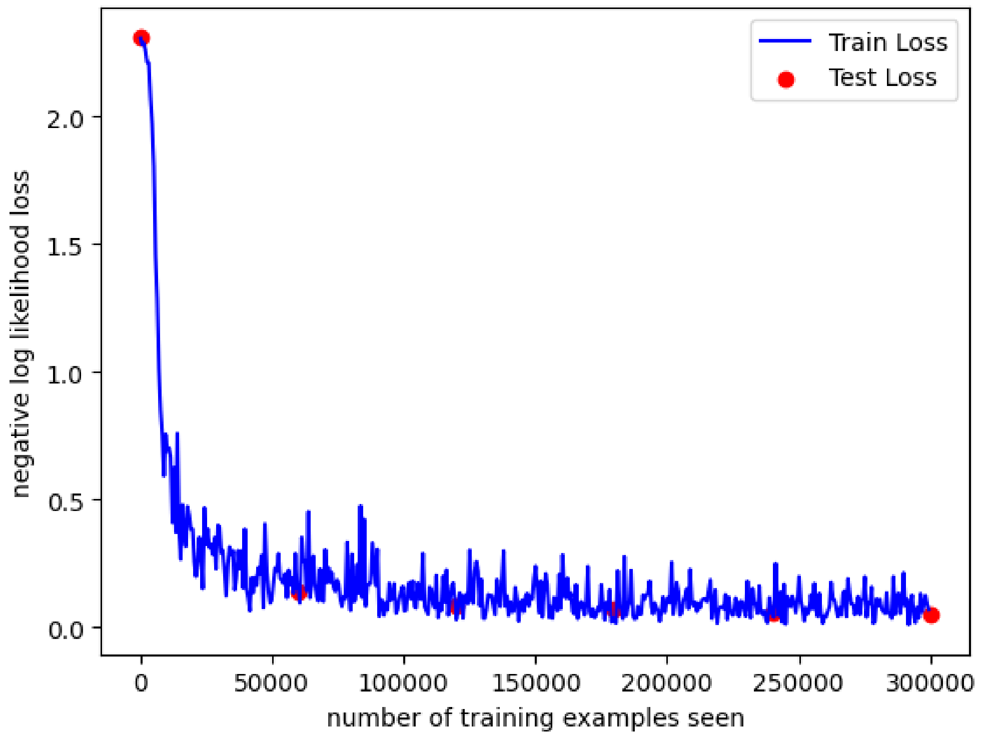


Figure 1C: A plot of the first six example digits from the test set

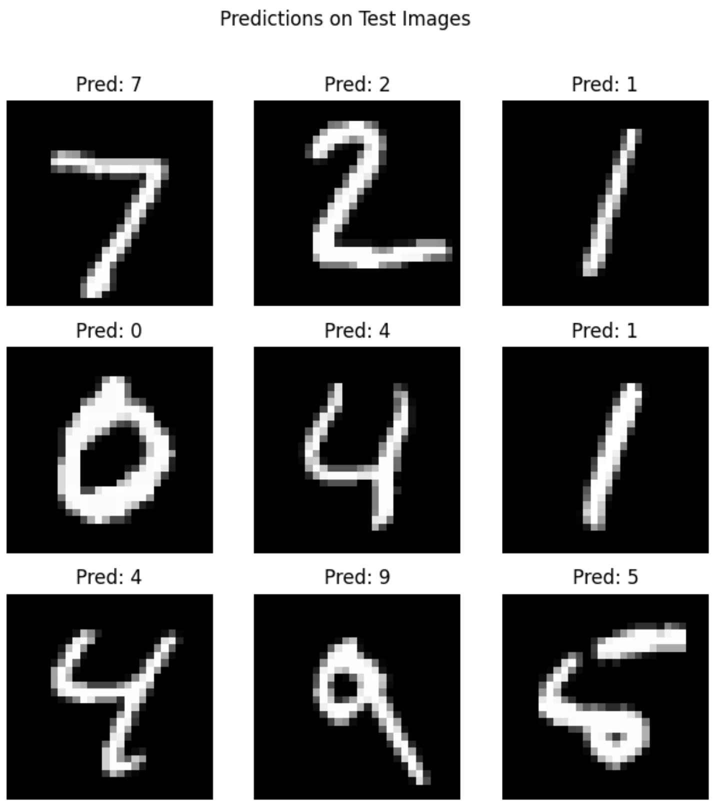
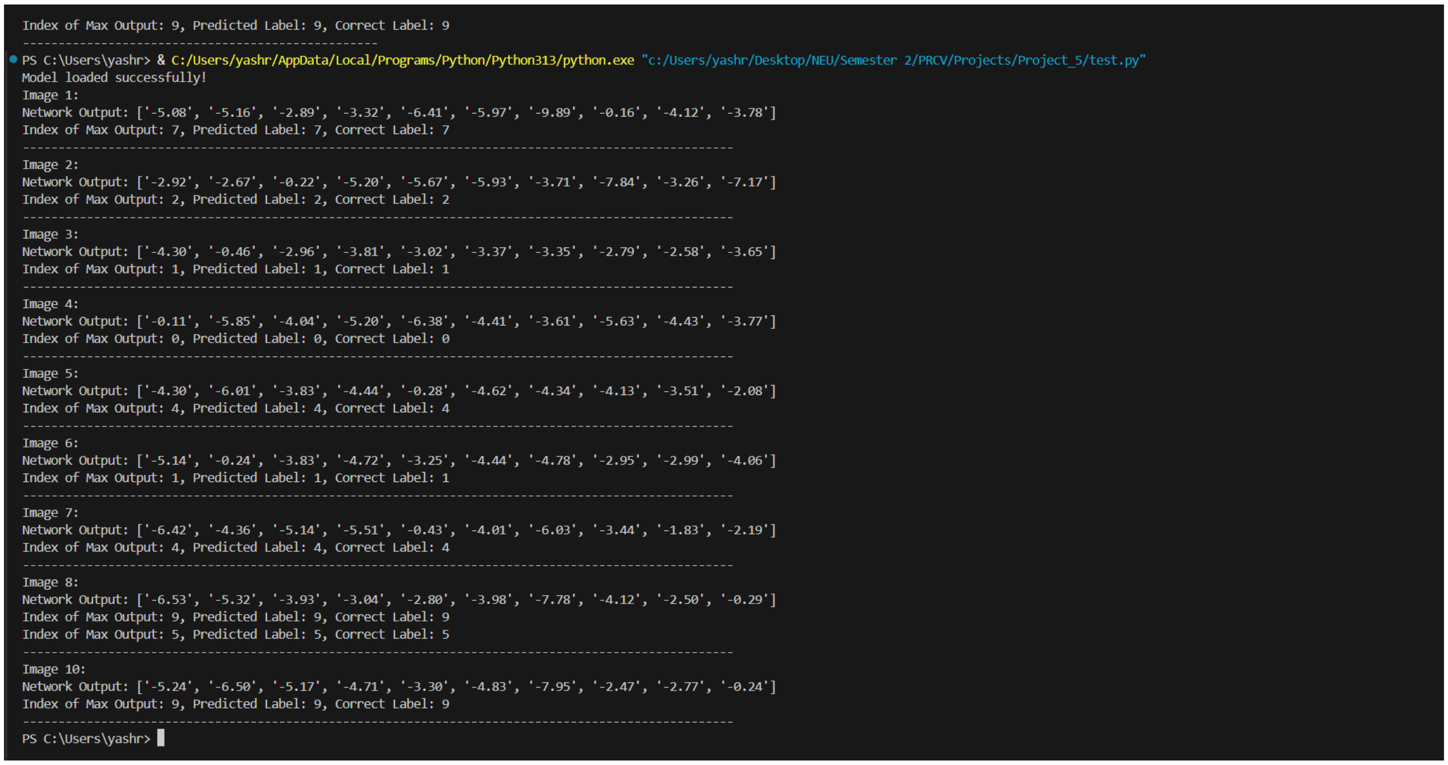


Figure 1D: Screen shot of your printed values and the plot of the first 9 digits of the test set



```
Index of Max Output: 9, Predicted Label: 9, Correct Label: 9
--------------------------------------------------
● PS C:\Users\yashr> & C:/Users/yashr/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/yashr/Desktop/NEU/Semester 2/PRCV/Projects/Project_5/test.py"
Model loaded successfully!
Image 1:
Network Output: ['-5.08', '-5.16', '-2.89', '-3.32', '-6.41', '-5.97', '-9.89', '-0.16', '-4.12', '-3.78']
Index of Max Output: 7, Predicted Label: 7, Correct Label: 7
--------------------------------------------------------------------------------------------------------
Image 2:
Network Output: ['-2.92', '-2.67', '-0.22', '-5.20', '-5.67', '-5.93', '-3.71', '-7.84', '-3.26', '-7.17']
Index of Max Output: 2, Predicted Label: 2, Correct Label: 2
--------------------------------------------------------------------------------------------------------
Image 3:
Network Output: ['-4.30', '-0.46', '-2.96', '-3.81', '-3.02', '-3.37', '-3.35', '-2.79', '-2.58', '-3.65']
Index of Max Output: 1, Predicted Label: 1, Correct Label: 1
--------------------------------------------------------------------------------------------------------
Image 4:
Network Output: ['-0.11', '-5.85', '-4.04', '-5.20', '-6.38', '-4.41', '-3.61', '-5.63', '-4.43', '-3.77']
Index of Max Output: 0, Predicted Label: 0, Correct Label: 0
--------------------------------------------------------------------------------------------------------
Image 5:
Network Output: ['-4.30', '-6.01', '-3.83', '-4.44', '-0.28', '-4.62', '-4.34', '-4.13', '-3.51', '-2.08']
Index of Max Output: 4, Predicted Label: 4, Correct Label: 4
--------------------------------------------------------------------------------------------------------
Image 6:
Network Output: ['-5.14', '-0.24', '-3.83', '-4.72', '-3.25', '-4.44', '-4.78', '-2.95', '-2.99', '-4.06']
Index of Max Output: 1, Predicted Label: 1, Correct Label: 1
--------------------------------------------------------------------------------------------------------
Image 7:
Network Output: ['-6.42', '-4.36', '-5.14', '-5.51', '-0.43', '-4.01', '-6.03', '-3.44', '-1.83', '-2.19']
Index of Max Output: 4, Predicted Label: 4, Correct Label: 4
--------------------------------------------------------------------------------------------------------
Image 8:
Network Output: ['-6.53', '-5.32', '-3.93', '-3.04', '-2.80', '-3.98', '-7.78', '-4.12', '-2.50', '-0.29']
Index of Max Output: 9, Predicted Label: 9, Correct Label: 9
Index of Max Output: 5, Predicted Label: 5, Correct Label: 5
--------------------------------------------------------------------------------------------------------
Image 10:
Network Output: ['-5.24', '-6.50', '-5.17', '-4.71', '-3.30', '-4.83', '-7.95', '-2.47', '-2.77', '-0.24']
Index of Max Output: 9, Predicted Label: 9, Correct Label: 9
--------------------------------------------------------------------------------------------------------
PS C:\Users\yashr> █
```

Figure 1 E:10 network output values printed on the command line

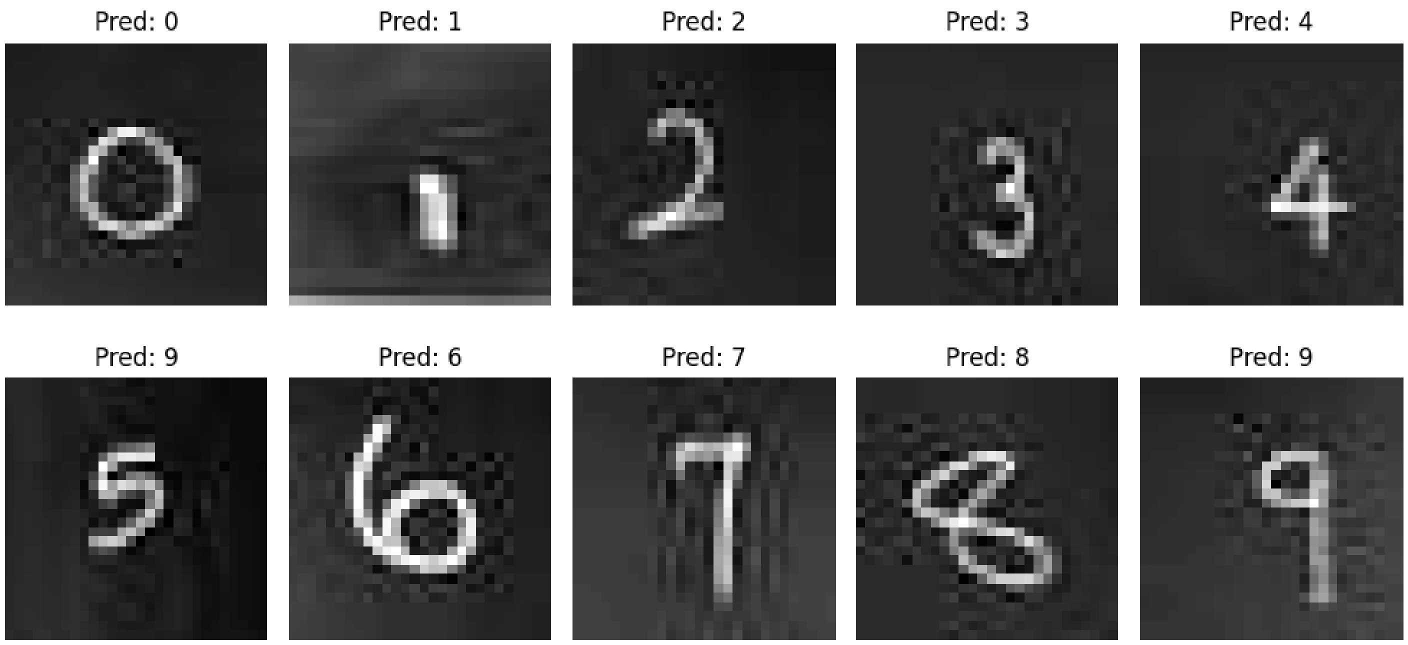


Figure 1 F: Results on Paint_Images (Handwritten dataset)

In conclusion, the network is well trained to accurately detect digits. As seen from the visualization above, it detects 9 out of 10 images correctly.

2.2 **Examine your network**

Objective: Acquire and visualize the MNIST dataset using torchvision and matplotlib to understand its characteristics. Design a convolutional neural network architecture according to specified layer configurations, including convolutional, pooling, and fully connected layers. Document the network's structure with a clear visual representation, such as a diagram. Emphasize the importance of data exploration and thoughtful network design in achieving optimal model performance. This task focuses on the essential steps of data handling and network architecture, setting the stage for effective model training.

Approach: The approach, as demonstrated in the provided code, involves loading the trained model, inspecting its weights, and visualizing the filters and their effects on input images.

1. Model Loading and Setup:
   - The trained model, saved in "model.pth," is loaded using torch.load() and its state dictionary is loaded into an instance of the Net class.
   - The network is set to evaluation mode using network.eval() to ensure consistent behavior.
   - The MNIST test dataset is loaded using torchvision.datasets.MNIST, with a transforms.ToTensor() transformation.
2. Weight Inspection and Printing:
   - The size of the first convolutional layer's weights (network.conv1.weight.size()) is printed, revealing the dimensions of the filter bank.
   - The individual weight matrices for each of the ten filters in the first convolutional layer are printed. This provides a numerical view of the filters.
3. Filter Visualization (Task 2A):
   - The weights of the first convolutional layer are visualized using matplotlib.pyplot.
   - A 3x4 grid of subplots is created, and each filter's weight matrix is displayed as a grayscale image using plt.imshow().
   - This visualization allows for a qualitative assessment of the filters, revealing patterns they are designed to detect.
4. Filter Application and Output Visualization (Task 2B):
   - The first image from the MNIST test dataset is retrieved and passed through the first convolutional layer using F.conv2d().
   - The output of this operation is visualized, showing the effect of each filter on the input image.
   - A 5x4 grid of subplots is created, displaying both the filter weights and the corresponding filtered image side-by-side.
   - This visualization helps to understand how each filter responds to the input image, revealing the features it extracts.
   - The gradient calculation is turned off using torch.no_grad() to prevent any unwanted changes to the trained weights during the filtering process.
5. Main Execution:
   - The main() function orchestrates the loading of the model and dataset, the inspection of weights, and the visualization of filters and their effects.

- The plot_taskA() and plot_taskB() functions handle the visualization aspects of the task.
- The code is designed to be executed from the command line, with the if __name__ == "__main__": guard ensuring that the main() function is called only when the script is run directly.
- The code is written in a modular way, with seperate functions to handle the different parts of the task.

Plots / Results:

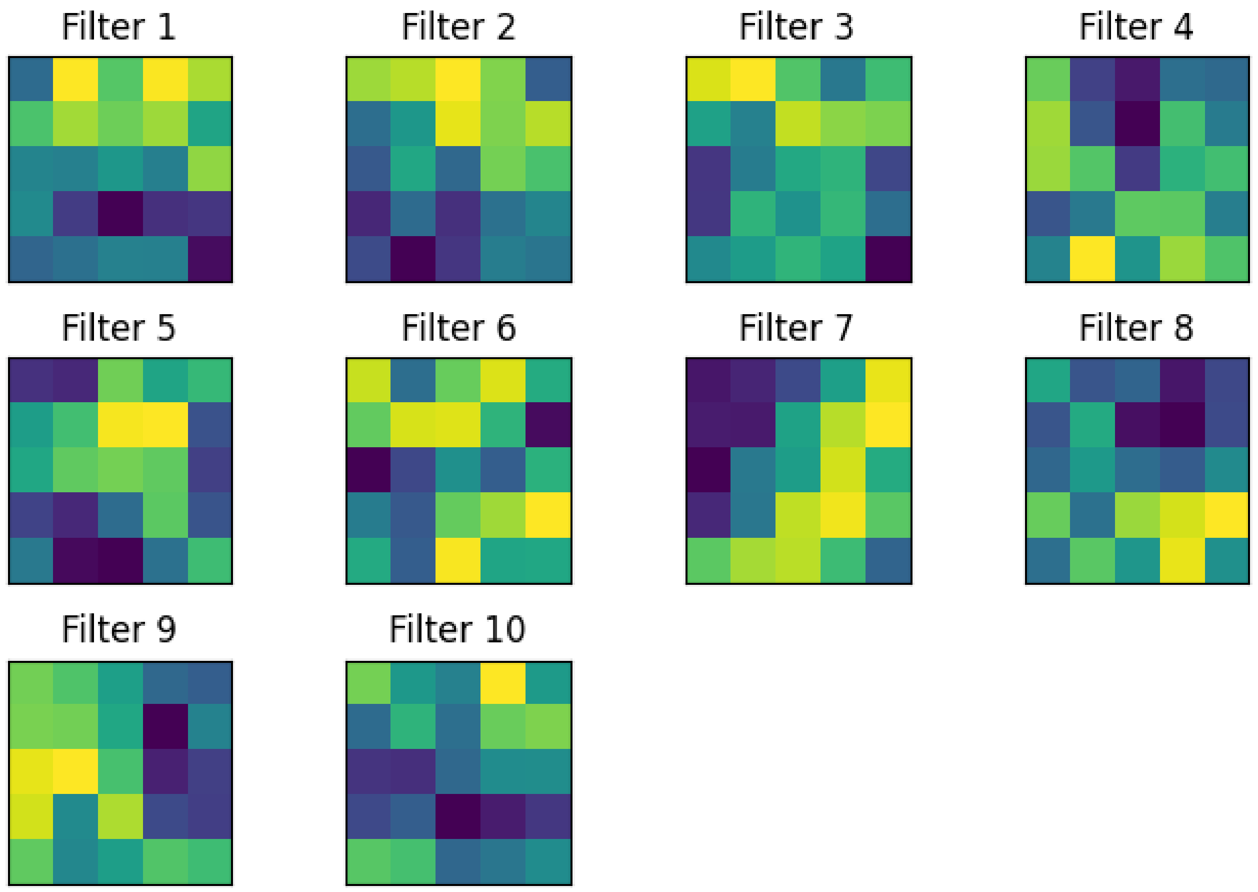


Figure 2A: Visualization of the ten filters

They don’t resemble standard, human-designed image processing filters.

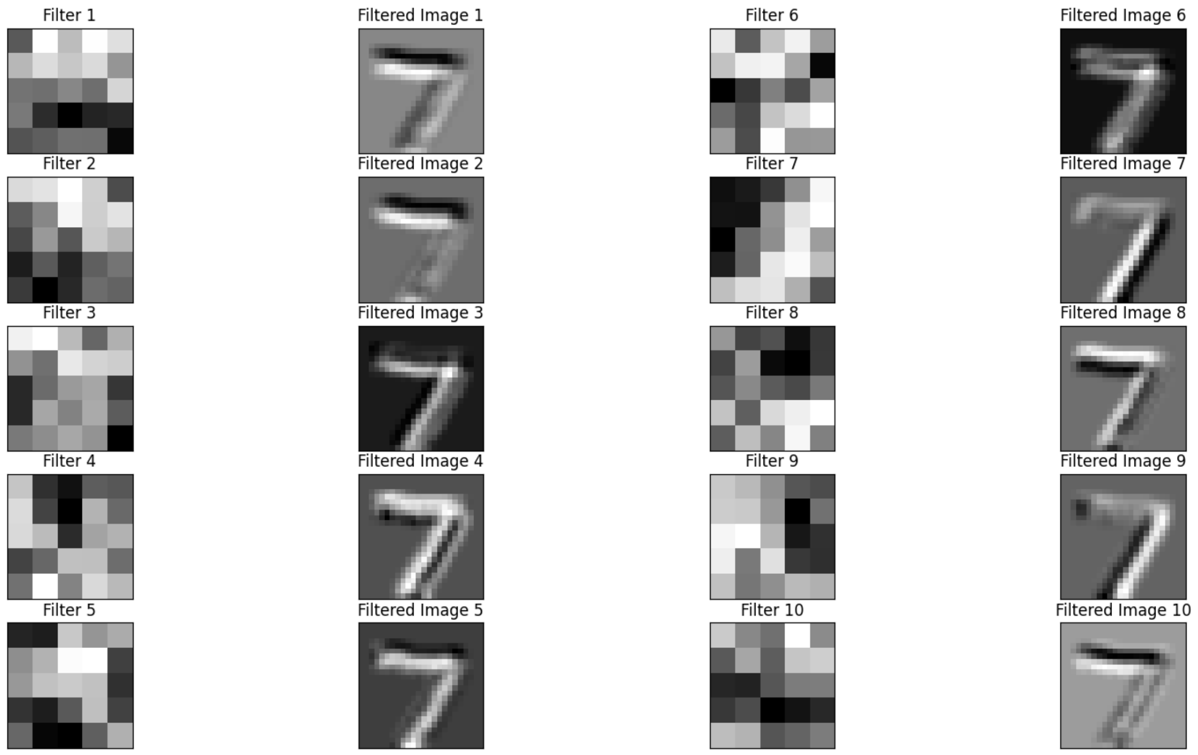


Figure 2B: Plot of the above 10 filters applied to a sample image

Even though the filters do not represent an standard filter, they seem to be working like them or combinations of the standard filters. Some of these filters are like Sobel filters applied at an angle along with Gaussian filters.

2.3 **Transfer Learning on Greek Letters**

Objective: The goal of this step is to re-use the the MNIST digit recognition network built in task 1 to recognize three different greek letters: alpha, beta, and gamma.

Approach: This task demonstrates transfer learning by adapting a pre-trained MNIST digit recognition network to classify Greek letters (alpha, beta, gamma). The approach involves loading the pre-trained model, freezing its weights, modifying the output layer for the new classification task, and training the modified model on the Greek letter dataset.

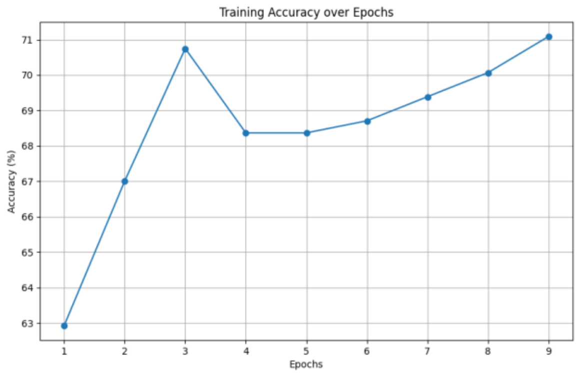
1. MNIST Model Loading and Preparation:

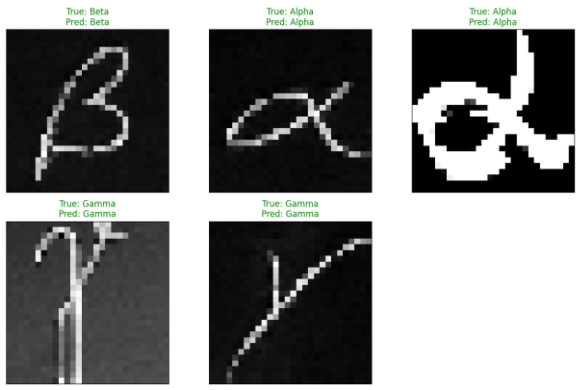
- The Net class, identical to the one used in Task 1, is defined to represent the MNIST network architecture.
- The pre-trained MNIST model weights are loaded from "model.pth" using torch.load() and applied to a new instance of the Net class.
- All parameters of the loaded network are frozen using param.requires_grad = False. This prevents the pre-trained weights from being updated during training, preserving the learned features.
- The final fully connected layer (network.fc2) is replaced with a new nn.Linear layer that outputs three classes, corresponding to alpha, beta, and gamma.
- The code prints the trainable parameters, confirming that only the new output layer's weights are being trained.

2. Greek Letter Dataset and Transformation:
   - The Greek letter dataset is loaded using torchvision.datasets.ImageFolder, which automatically organizes images from subdirectories (alpha, beta, gamma).
   - A custom GreekTransform class is defined to preprocess the Greek letter images:
     - It converts RGB images to grayscale using torchvision.transforms.functional.rgb_to_grayscale.
     - It then resizes the image to 28x28, to match the size of the MNIST images, instead of cropping.
     - Finally, it inverts the image intensities using torchvision.transforms.functional.invert to match the MNIST digit format (white digits on a black background).
   - A DataLoader is created for the Greek dataset, applying the GreekTransform and normalization (using MNIST mean and standard deviation).
   - The visualize_greek_examples() function is used to display a few example images from the loaded dataset, ensuring the data is loaded correctly.
3. Training the Modified Model:
   - An optim.SGD optimizer is created, but it only optimizes the parameters of the newly added output layer (the frozen layers are not optimized).
   - The train() function implements the training loop:
     - It iterates through the Greek letter dataset, performs forward and backward passes, and updates the output layer's weights.
     - The F.cross_entropy loss function is used for multi-class classification.
     - It calculates and displays the training accuracy.
   - The test() function evaluates the trained model on the Greek letter dataset, calculating and displaying the accuracy.
4. Prediction and Visualization:
   - The visualize_model_predictions() function displays a few example images from the loaded dataset, and shows the predicted labels, and the true labels, with the color of the label text being green if the prediction was correct, and red if it was wrong.
   - The predict_custom_image() function is created to allow the user to predict the class of custom greek letter images.
   - The trained model's state dictionary is saved to "greek_model.pth" for later use.
   - The training accuracy is plotted against the number of epochs to visualize the training progress.
5. Epoch Selection and Custom Image Testing:
   - The code trains the model for a specified number of epochs (adjustable). The number of epochs needed to achieve reasonable accuracy is determined empirically by observing the training progress.
   - The user is expected to capture their own images of alpha, beta, and gamma symbols, preprocess them (crop, resize, etc.), and use the predict_custom_image() function to test the model's performance on these custom images.
   - The code provides the tools to test the trained model on custom images.
   - The code displays the class prediction, and the confidence of the prediction.

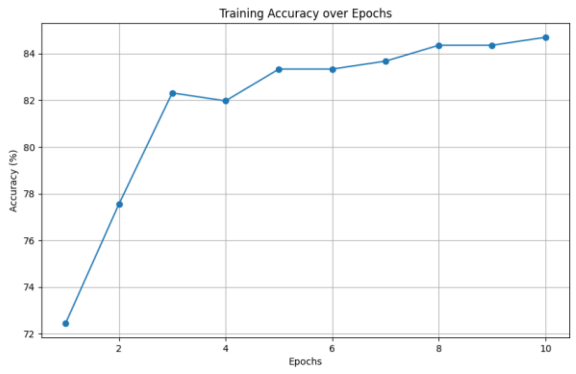
<u>Results / Plots</u>:

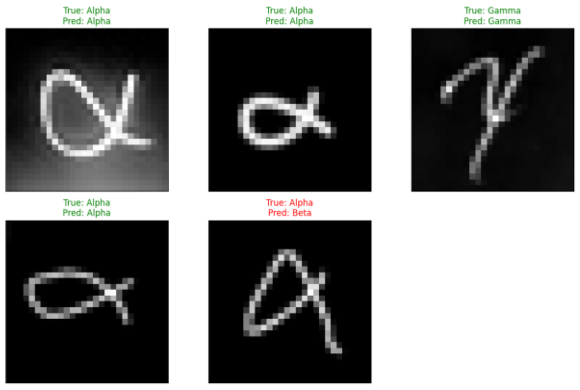
Below are the plots of training accuracy over the varying epochs values and their visualization of the predictions v/s true value for random images from the Greek_dataset.



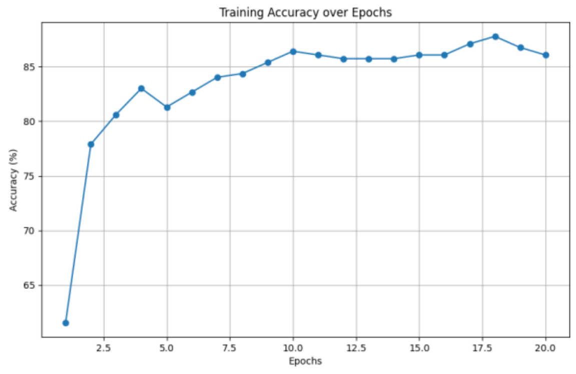


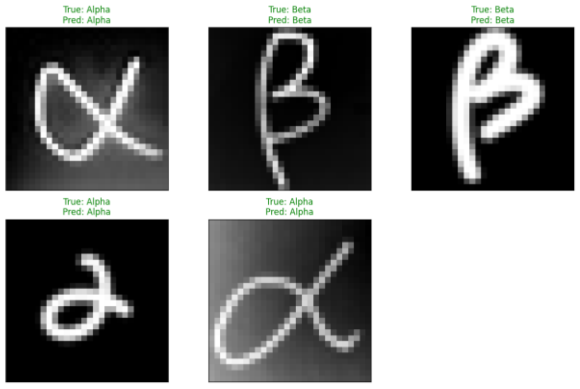
9 Epochs – All 5 predictions are correct



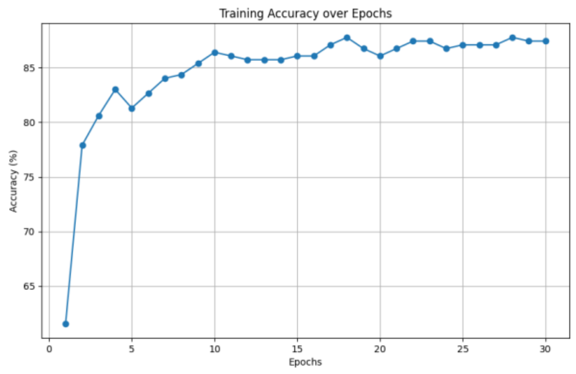


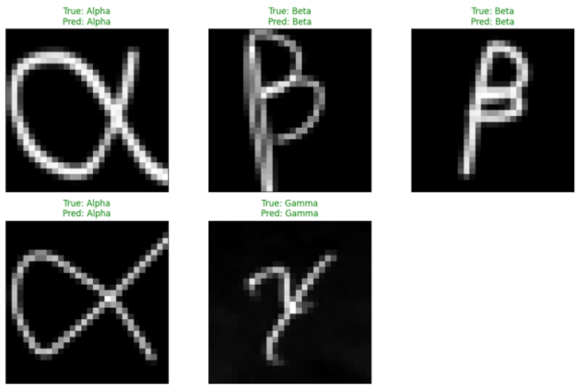
10 Epochs – 4/5 predictions are correct





20 Epochs – 5/5 predictions are correct





30 Epochs – 5/5 predictions are correct

2.4 Design your own experiment

Objective: The final main task is to undertake some experimentation with the deep network for the MNIST task. The goal is to evaluate the effect of changing different aspects of the network. Pick at least three dimensions along which you can change the network architecture and see if you can optimize the network performance and/or training time along those three dimensions. Automate the process.

Approach:

Plan-

The plan for automating wide range of experiments is by varying hyper-parameters as under:

# Total = 4 x 1 x 4 x 4 = 64 Network variations

conv_layers = [2, 3, 4, 5]

filter_sizes = [3]

batch_sizes = [32, 64, 128, 256]

epochs_list = [5, 10, 15, 20]

Hypothesis-

The impact of varying convolutional layers, filter sizes, batch sizes, and training epochs on the Fashion-MNIST dataset, here are my hypotheses:

1. Number of Convolutional Layers:
   - Hypothesis: Increasing the number of convolutional layers will initially improve accuracy, as the network will learn more complex features. However, beyond a certain point, adding more layers will lead to diminishing returns, increased training time, and potential overfitting.
   - Reasoning: Deeper networks have a greater capacity to learn intricate patterns. However, they also introduce more parameters, which can make training more difficult and prone to overfitting if not properly regularized.
2. Filter Size:
   - Hypothesis: Using a filter size of 3x3 will yield better results than larger filter sizes.

- Reasoning: 3x3 filters are common because they are effective at capturing local features while maintaining a reasonable number of parameters. Larger filters will increase the computational cost, and are not needed to capture the features of the fashion MNIST dataset.

3. Batch Size:

- Hypothesis: Larger batch sizes will lead to faster training times per epoch but may require more epochs to converge. Smaller batch sizes will result in slower training but potentially better generalization due to the noisy gradient updates.
- Reasoning: Larger batch sizes provide more stable gradient estimates, which can speed up training. However, they might miss finer details in the data. Smaller batch sizes introduce more stochasticity, which can act as a form of regularization.

4. Number of Epochs:

- Hypothesis: Increasing the number of epochs will improve accuracy up to a point, after which the network will begin to overfit, and the accuracy will start to decrease.
- Reasoning: More epochs allow the network to learn more from the data. However, at a certain point the network will start to memorize the training data, hurting the generalization of the model.

<u>Plots</u>:

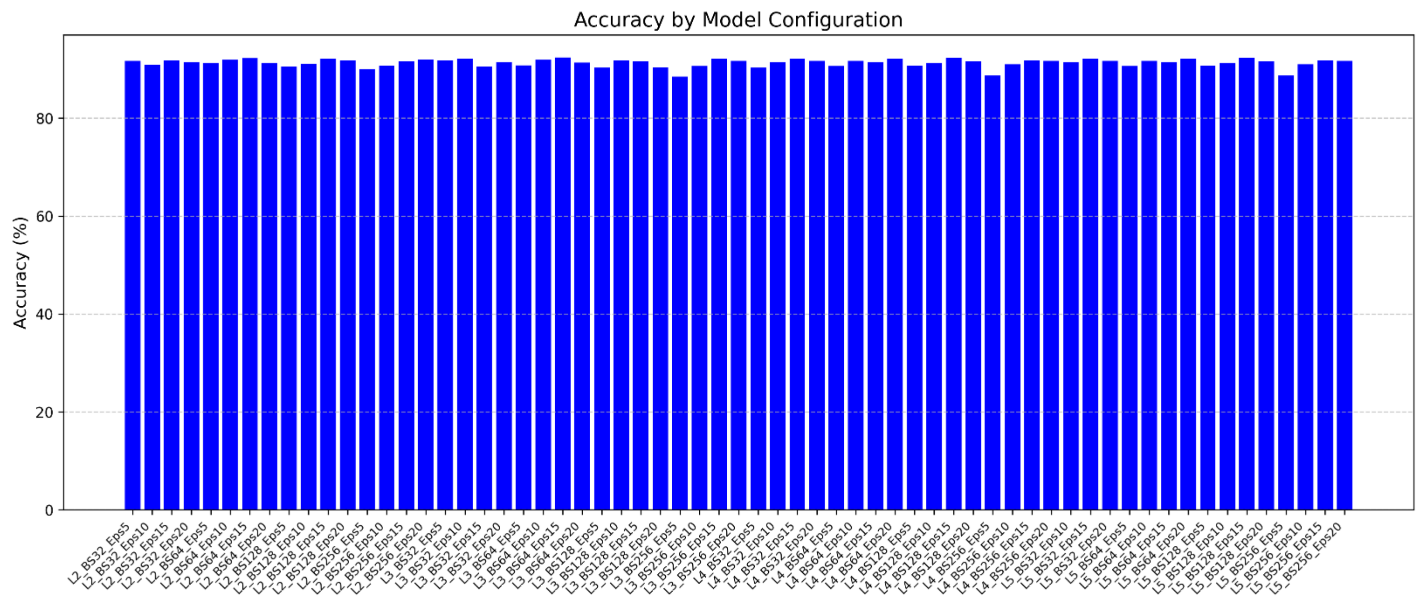


Figure 4A: Accuracy comparisons

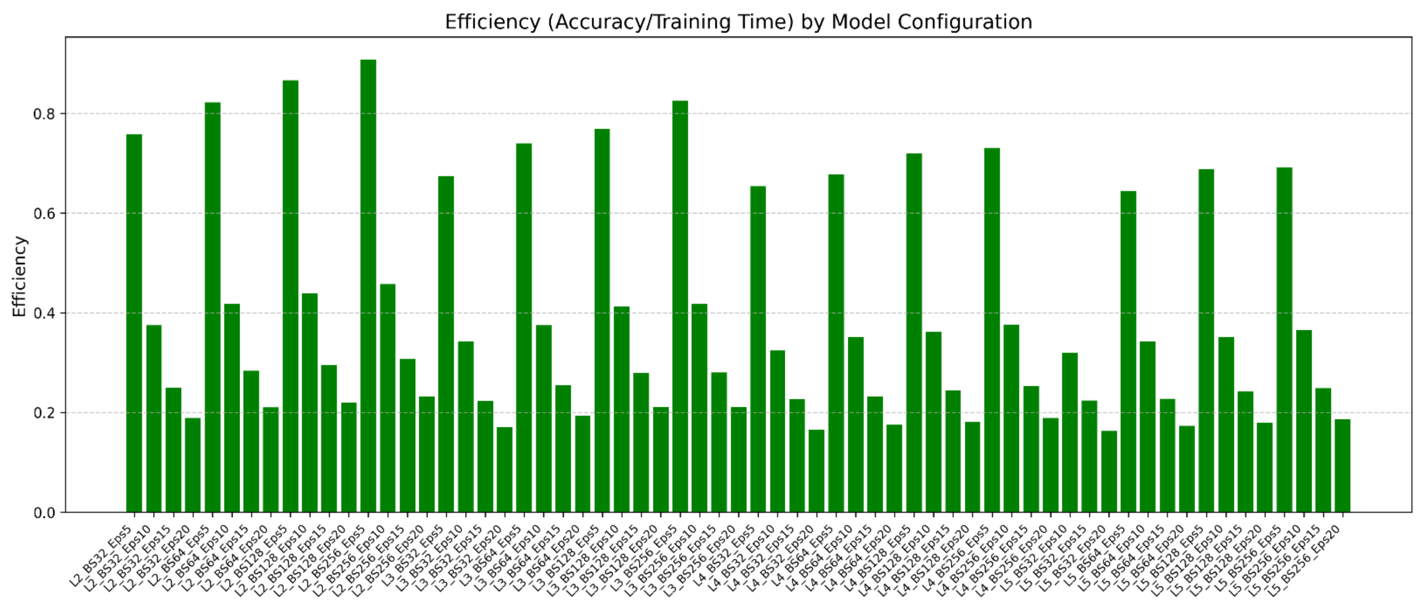


Figure 4B: Efficiency comparisons

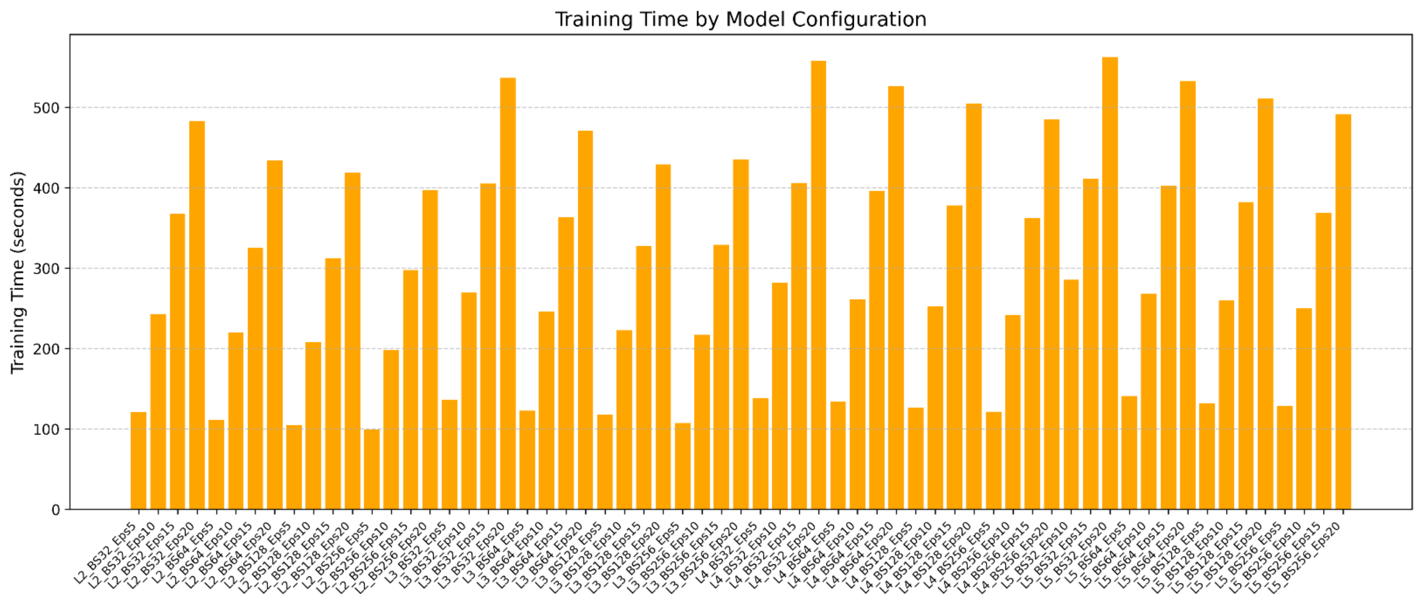


Figure 4C: Training time comparisons

<u>*Conclusions based on plots*</u>:

Analysis of the Accuracy Plot

1. High Accuracy Overall:
   - The most striking feature is that the accuracy is generally very high across all configurations, consistently above 80% and often exceeding 90%. This indicates that the Fashion-MNIST dataset, while more challenging than MNIST, can be learned effectively with a variety of CNN architectures.

2. Limited Variation:
   - There's relatively little variation in accuracy between different configurations. This suggests that the network is quite robust to changes in the number of convolutional layers, batch size, and epochs, within the ranges tested.
3. No Clear Trends:
   - It's challenging to identify clear trends related to specific hyperparameters. For example, there's no obvious pattern showing that increasing the number of convolutional layers consistently improves or worsens accuracy. Similarly, variations in batch size and epochs don't seem to have a dramatic impact.

Comparison with Hypotheses

1. Number of Convolutional Layers:
   - Hypothesis: We expected an initial improvement in accuracy with more layers, followed by diminishing returns and potential overfitting.
   - Reality: The plot doesn't show a clear trend. Accuracy remains relatively stable across different numbers of layers. This might be because the Fashion-MNIST dataset doesn't require extremely deep networks to achieve high performance. Or, it could be that the range we tested was not wide enough to see the effect. We only tested from 2 to 5 convolutional layers.
2. Filter Size:
   - Hypothesis: We hypothesized that a 3x3 filter size would be effective and that larger filters would not provide significant benefits.
   - Reality: Since the filter size was held constant at 3x3 in this experiment, we cannot directly evaluate this hypothesis. However, the high accuracy achieved suggests that 3x3 filters are indeed sufficient for this task.
3. Batch Size:
   - Hypothesis: We expected larger batch sizes to lead to faster training but potentially lower generalization.
   - Reality: The accuracy plot doesn't reveal a clear relationship between batch size and accuracy. However, we would need to look at the training time plot to see if larger batch sizes resulted in faster training, as hypothesized.
4. Number of Epochs:
   - Hypothesis: We expected accuracy to improve with more epochs up to a point, followed by overfitting.
   - Reality: Again, the accuracy plot doesn't show a strong correlation between the number of epochs and accuracy. This could be because the number of epochs tested was not high enough to cause overfitting, or because the model generalizes well even with fewer epochs.

Possible Explanations for the Discrepancies

- Dataset Complexity: The Fashion-MNIST dataset might not be complex enough to reveal significant differences in performance across the tested hyperparameter ranges.
- Hyperparameter Ranges: The ranges of hyperparameters we explored might have been too narrow. A wider range might have shown more pronounced effects.
- Optimization Robustness: The Adam optimizer used in the code is known for its robustness, which might have minimized the impact of hyperparameter variations.

- Evaluation Metric: Accuracy alone might not capture subtle differences in model performance. Other metrics, such as precision, recall, F1-score, or the confusion matrix, might provide more insights.

Analysis of the Efficiency Plot

1. Significant Variation:
   - Unlike the accuracy plot, the efficiency plot shows substantial variation across different configurations. This indicates that some models achieve much better efficiency than others.
2. Identifying Efficient Configurations:
   - Several configurations stand out with high efficiency. These are the models that achieve good accuracy with relatively low training times.
   - Look for the tallest bars in the plot to identify these efficient configurations.
3. Identifying Inefficient Configurations:
   - Conversely, some configurations have very low efficiency. These models either have low accuracy or take a long time to train, or both.
   - Look for the shortest bars in the plot to identify these inefficient configurations.
4. Trends and Patterns:
   - Look for any trends or patterns in the plot. For example, do models with certain hyperparameters consistently have higher or lower efficiency?
   - Do models with larger batch sizes have higher efficiency?
   - Do models with fewer convolutional layers have higher efficiency?
   - Do models with fewer epochs have higher efficiency?

Comparison with Hypotheses and Accuracy Plot

1. Batch Size:
   - Hypothesis: We hypothesized that larger batch sizes would lead to faster training but potentially lower generalization.
   - Reality: The efficiency plot, combined with the accuracy plot, can help us evaluate this. If models with larger batch sizes have higher efficiency, it would support this hypothesis.
2. Number of Convolutional Layers:
   - Hypothesis: We expected an initial improvement in accuracy with more layers, followed by diminishing returns and potential overfitting.
   - Reality: The efficiency plot can help us see if models with fewer convolutional layers have higher efficiency, indicating that they achieve good accuracy with faster training.
3. Number of Epochs:
   - Hypothesis: We expected accuracy to improve with more epochs up to a point, followed by overfitting.
   - Reality: The efficiency plot can help us see if models with fewer epochs have higher efficiency, indicating that they achieve good accuracy with faster training.

Possible Explanations for the Observed Trends

- Training Time Dominance: The efficiency metric is heavily influenced by training time. If some models take significantly longer to train, they will have lower efficiency, even if their accuracy is slightly higher.
- Batch Size Impact: Larger batch sizes might lead to faster training due to better hardware utilization, but might not significantly impact accuracy.
- Model Complexity: Simpler models (fewer layers, fewer epochs) might train faster and achieve comparable accuracy, resulting in higher efficiency.

Conclusions:

Top 5 Models by Accuracy:

Model	Accuracy	Training Time	Efficiency
L3_BS64_Eps15	92.36	363.588442	0.254023
L5_BS128_Eps15	92.34	382.060587	0.241689
L4_BS128_Eps15	92.34	378.060587	0.244247
L2_BS64_Eps15	92.24	325.197030	0.283643
L3_BS32_Eps10	92.16	269.623045	0.341811

Top 5 Models by Efficiency:

Model	Accuracy	Training Time	Efficiency
L2_BS256_Eps5	89.94	99.117623	0.907407
L2_BS128_Eps5	90.54	104.596752	0.865610
L3_BS256_Eps5	88.51	107.274470	0.825080
L2_BS64_Eps5	91.27	111.091179	0.821577
L3_BS128_Eps5	90.26	117.362179	0.769072

3. Extensions:

3.1. Extension 1: Extended Task 4 to a different dataset and assessing change in other parameters.

It designed to evaluate the performance of a convolutional neural network (CNN) on the MNIST dataset, focusing on how different hyperparameter configurations affect the model's efficiency. The core function is to train and test a CNN with varying settings, specifically adjusting the number of convolutional layers, filter sizes, dropout rates, batch sizes, and the number of training epochs. The code iterates through a predefined set of these hyperparameters, creating a unique model and training scenario for each combination. For each run, it records the training time and the final accuracy achieved on the test dataset.

The heart of the experiment lies in the experiment function, which handles the data loading, model initialization, training, and evaluation processes. It utilizes PyTorch's DataLoader to efficiently manage the MNIST dataset, applying necessary transformations like converting images to tensors and normalizing them. The BasicNetwork class defines the CNN architecture, which is then instantiated with the current hyperparameter settings. During training, the code calculates the negative log-likelihood loss and uses the Stochastic Gradient Descent (SGD) optimizer to update the model's weights. The training progress, including loss and accuracy, is tracked and potentially visualized through plots, saved as PNG files.

Finally, the main function orchestrates the entire experiment by looping through all specified hyperparameter combinations. It calls the experiment function for each set of parameters and stores the results, including training time and accuracy, in a structured format. This allows for a comprehensive comparison of how different configurations impact the model's performance. The results are then printed, summarizing the effectiveness of each hyperparameter combination. This systematic approach enables the user to identify the most efficient model settings, balancing accuracy with training time, for the given task.

Code explanation:

1. Imports and Setup:

- Libraries:
  - torch: PyTorch library for tensor computations and neural networks.
  - torchvision: Datasets, models, and transforms for computer vision.
  - torch.nn: Neural network modules.
  - torch.optim: Optimization algorithms.
  - torch.nn.functional as F: Functional interface for neural network operations.
  - matplotlib.pyplot as plt: Plotting library.
  - os: Operating system interface for creating directories.
- Directory Creation:
  - os.makedirs('results', exist_ok=True): Creates a directory named "results" to store saved models and optimizer states. exist_ok=True prevents errors if the directory already exists.
- Hyperparameters:
  - N_EPOCHS, LEARNING_RATE, MOMENTUM, LOG_INTERVAL, batch_size: These are the hyperparameters for training the network.

2. Network Definition (BasicNetwork class):

- Architecture:
  - The network has a configurable number of convolutional layers.
  - It starts with two convolutional layers (conv1, conv2), followed by an optional number of additional convolutional layers (conv).
  - Max pooling (F.max_pool2d) is used after the first two convolutional layers.
  - Dropout (nn.Dropout2d) is used for regularization.
  - Fully connected layers (fc1, fc2) are used for final classification.
- get_fc1_input_size():
  - Calculates the input size for the first fully connected layer (fc1) based on the output size of the convolutional layers.
- forward():

- Defines the forward pass of the network, specifying the order of operations.

3. Training and Testing Functions:

- train():
  - Trains the model for one epoch.
  - Calculates the loss using F.nll_loss.
  - Performs backpropagation and updates the model's weights using the optimizer.
  - Prints training progress and saves the model and optimizer states periodically.
- test():
  - Evaluates the model on the test dataset.
  - Calculates the test loss and accuracy.
  - Prints the test results.
- plot_curve():
  - Plots the training and test losses over epochs.
  - Saves the plot to a file.

4. Experiment Function (experiment()):

- Data Loading:
  - Loads the MNIST dataset using torchvision.datasets.MNIST and creates data loaders for training and testing.
- Model and Optimizer Initialization:
  - Creates an instance of the BasicNetwork class and initializes the optimizer.
- Training and Evaluation Loop:
  - Trains the model for the specified number of epochs.
  - Evaluates the model after each epoch.
  - Plots the training and test losses.
- Returns:
  - Returns the initial accuracy of the model on the test set.

5. Main Function (main()):

- Hyperparameter Grid Search:
  - Iterates through different combinations of hyperparameters (number of epochs, batch size, number of layers, convolutional filter size, dropout rate).
- Experiment Execution:
  - Calls the experiment() function for each hyperparameter combination.
  - Stores the results in a list of dictionaries.
- Error Handling:

    - Handles potential errors during the experiment.
- Results Printing:
    - Prints a summary of the experiment results.

Key Features:

- Hyperparameter Tuning: The code systematically explores the impact of various hyperparameters on the model's performance.
- Modular Design: The code is organized into functions and classes, making it easy to modify and extend.
- Error Handling: The code includes error handling to prevent the experiment from crashing.
- Results Tracking: The code stores the results of each experiment in a dictionary, making it easy to analyze the impact of different hyperparameters.
- Visualization: The code generates plots of the training and test losses, providing insights into the model's learning process.

In summary, this code provides a framework for conducting experiments to optimize the performance of a CNN on the MNIST dataset.

3.2. Extension 2: Changing hyperparameters for MNIST dataset to optimize the CNN.

This Python code is designed to conduct a series of experiments on the MNIST dataset, evaluating the impact of different hyperparameters on the performance and training time of a convolutional neural network (CNN). The script defines a flexible CNN architecture, DynamicCNN, that can be configured with varying numbers of convolutional layers and filter sizes. It then systematically trains and evaluates this model across a grid of hyperparameter combinations, including different batch sizes and training epochs. The primary goal is to understand how these parameters affect the model's accuracy and the time it takes to train, ultimately aiming to identify the most efficient configurations for the MNIST digit classification task.

The train_and_evaluate function is the workhorse of this script, handling the training and testing of the CNN for each hyperparameter set. It loads the MNIST dataset, sets up the model and optimizer (Adam), and then iterates through the specified number of training epochs. During training, it calculates the loss, performs backpropagation, and updates the model's weights. It also tracks the training time and evaluates the model's accuracy on the test set. The main function orchestrates the entire experiment, looping through different combinations of convolutional layers, filter sizes, batch sizes, and epochs. It saves the trained models and their performance metrics, along with the training times, to a CSV file for later analysis.

In essence, this code automates the process of training and evaluating multiple CNN models with different configurations on the MNIST dataset. By systematically varying the hyperparameters and recording the results, the script enables the user to analyze the trade-offs between model complexity, training time, and accuracy. This allows for the identification of optimal hyperparameter settings for efficient and effective digit classification, highlighting the impact of architectural choices and training parameters on the model's overall performance.

Step-by-step code explanation:

1. Importing Libraries:
    - Imports necessary PyTorch modules (torch, torch.nn, torch.optim, torchvision), along with time, pandas, and os.

2. DynamicCNN Class:
   - Defines a configurable CNN model.
   - __init__: Initializes convolutional layers based on num_conv_layers and filter_size.
   - _get_conv_output_size: Calculates the input size for the fully connected layer.
   - forward: Defines the forward pass of the network.
3. train_and_evaluate Function:
   - Loads the MNIST dataset using torchvision.datasets.MNIST.
   - Initializes the DynamicCNN model, loss function (nn.CrossEntropyLoss), and optimizer (optim.Adam).
   - Trains the model for num_epochs, tracking training loss and time.
   - Evaluates the model on the test set and calculates accuracy.
   - Saves the trained model.
   - Returns training time, accuracy, and the trained model.
4. main Function:
   - Sets up the device (CPU).
   - Defines hyperparameter ranges to explore.
   - Iterates through hyperparameter combinations:
     - Trains and evaluates the model using train_and_evaluate.
     - Handles potential RuntimeError during training.
     - Saves the trained model when the batch size changes.
     - Tracks results (layers, filter size, batch size, epochs, training time, accuracy).
   - Handles KeyboardInterrupt for user interruption.
   - Saves the results to a CSV file.
5. Execution (if __name__ == "__main__":)
   - Calls the main function to start the experiment.

What it does and how it achieves it:

This code automates the hyperparameter tuning of a CNN for MNIST digit classification. It achieves this by:

- Defining a flexible CNN architecture (DynamicCNN).
- Systematically exploring a grid of hyperparameter combinations.
- Training and evaluating the model for each combination.
- Tracking and saving the results for analysis.

This allows the user to determine the optimal hyperparameters for the best performance and efficiency on the MNIST dataset.

Plots:

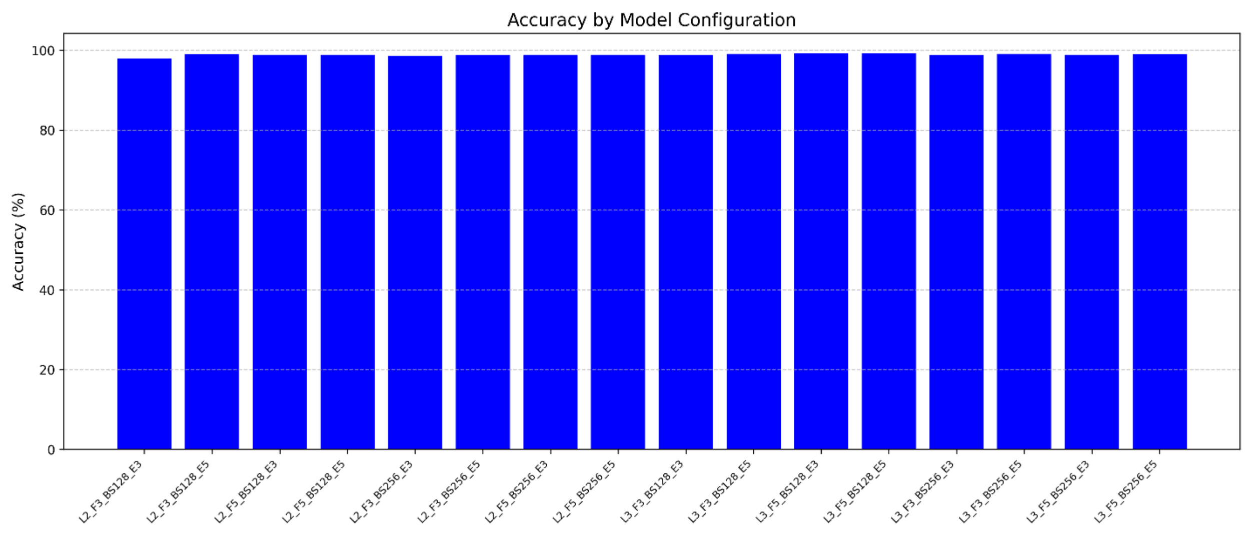


Figure 3.2 A: Accuracy Plot

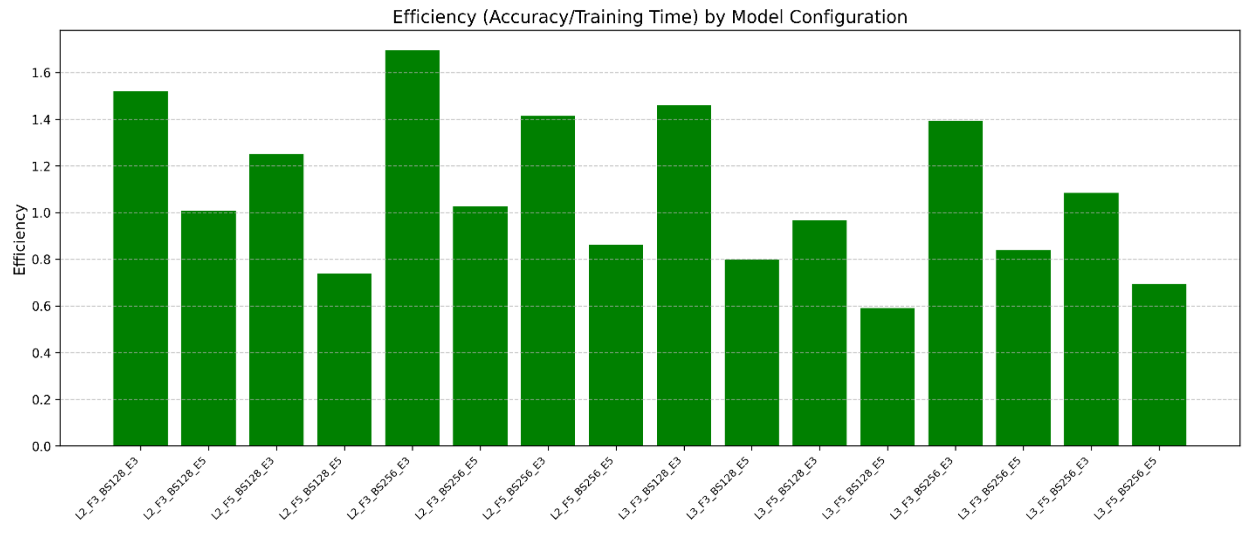


Figure 3.2 B: Efficiency Plot

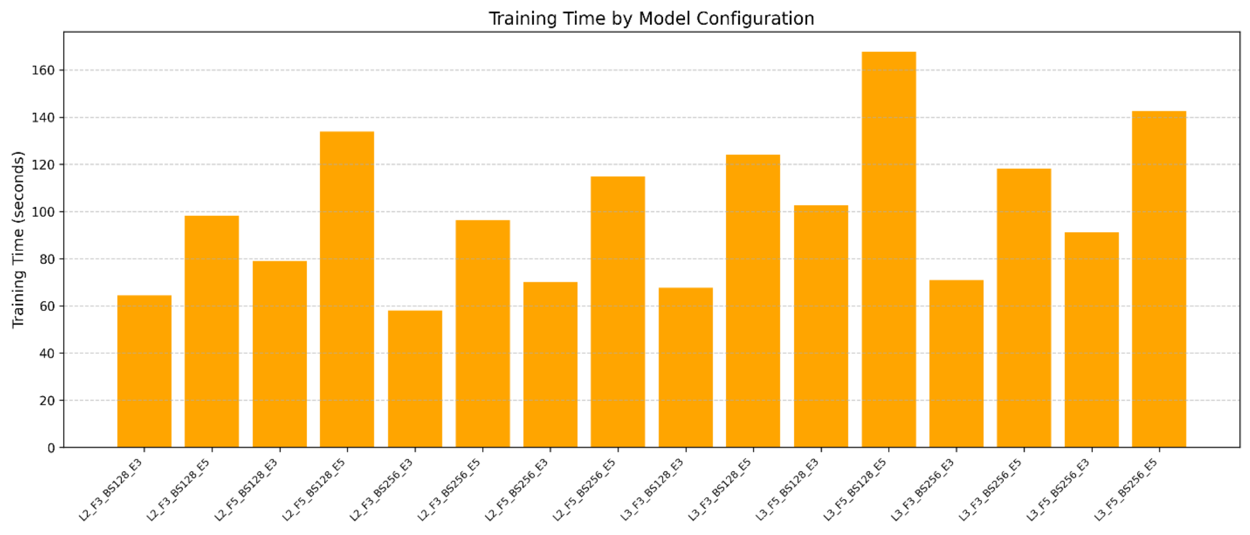


Figure 3.2 C: Training Time Plot

Results:

Top 5 Models by Accuracy:

Model	Accuracy	Training Time	Efficiency
L3_F5_BS128_E3	99.30	102.610468	0.967738
L3_F5_BS128_E5	99.16	167.769052	0.591051
L3_F3_BS128_E5	99.12	124.088160	0.798787
L3_F3_BS256_E5	99.09	118.285268	0.837721
L3_F5_BS256_E5	99.01	142.519451	0.694712

Top 5 Models by Efficiency:

Model	Accuracy	Training Time	Efficiency
L2_F3_BS256_E3	98.55	58.097764	1.696279
L2_F3_BS128_E3	97.95	64.413940	1.520634
L3_F3_BS128_E3	98.83	67.636728	1.461188
L2_F5_BS256_E3	98.87	69.910538	1.414236
L3_F3_BS256_E3	98.83	71.005771	1.391859