ECSE 551 Mini-Project 3: Using Machine Learning for Image Classification

Yeo Kiah Huah
McGill University
ID: 261252535

Kaitlyn Pereira
McGill University
ID: 261023292

Melody Wang
McGill University
ID: 261053910

Abstract

In this project, we explore supervised learning techniques to classify images containing digits from an unknown language alongside characters from the QMNIST dataset. The task is to develop an accurate image classification model capable of distinguishing among ten distinct classes (0-9). Our approach involves training deep learning models using the provided labeled dataset (Train.pkl and Train_labels.csv) and predicting the labels for unseen data (Test.pkl). We experiment with different convolutional neural network architectures and regularization techniques to improve generalization performance. Our final model achieves a validation accuracy of 98.42% and a test score of 98.83% on the Kaggle competition. We observe that proper model architecture design and learning rate experimentation can significantly affect model performance.

1 Introduction

Handwritten digit classification is a well-studied problem in computer vision. In this project, we are tasked with classifying images containing digits from an unknown language along with QMNIST characters. Each image contains one target digit from the unknown language, which we are required to identify among ten possible classes. This mini-project presents an opportunity to develop and evaluate a machine learning pipeline for supervised image classification using Python and PyTorch.

The dataset presents challenges such as mixed character presence, potentially noisy backgrounds, and unknown digit styles, requiring the model to learn robust, discriminative features. To tackle this, we explore several neural network architectures, namely convolutional neural networks (CNNs), known for their success in visual recognition tasks. We also investigate various training techniques like batch normalization and dropout to enhance generalization.

Our final solution is selected based on validation accuracy and leaderboard performance. Throughout this report, we discuss our design decisions, evaluate the strengths and weaknesses of different models, and provide insights into what contributed to our best-performing solution.

2 Dataset

The dataset for this project consists of:

- Train.pkl: A Python pickle file containing the training images.
- Train_labels.csv: A CSV file with labels (integers 0–9) corresponding to each training image.
- Test.pkl: A pickle file containing unlabeled test images for which we must generate predictions.

Preprint. Under review.

Each image is grayscale, containing one digit from an unknown script, possibly placed amidst or alongside QMNIST-style characters. Therefore, this is a classification problem with moderate noise and possible background clutter.

The images contained in the pickle file were already standardized and in the correct size. As the images themselves were fairly simple, we determined there was no need for additional preprocessing steps.

The distribution of the classes (0-9) can be seen in the Appendix Fig. 1. While the distribution is not perfectly balanced, the highest represented class only appears 1.24 times more than the lowest represented class. We believe this is not enough to cause issues with class imbalances, and therefore there is no need for under- or over- sampling.

3 Proposed Approach

3.1 Baseline CNN

The first model we implemented is a simple baseline CNN. It consists of two convolutional layers followed by max pooling, and two fully connected layers, with ReLU activation and dropout regularization. The first convolutional layer applies 32 filters of size 5×5 to the input image (28×28), reducing the spatial dimensions after the convolution and subsequent 2×2 max pooling. The second convolutional layer doubles the number of filters to 64, again followed by 2×2 max pooling. This is followed by flattening the output and passing it through a dense layer of 512 units with ReLU activation and a dropout layer (rate 0.5) to mitigate overfitting. Finally, the output layer produces class scores via a log-softmax function. This model served as a foundational benchmark to evaluate the performance improvements introduced by deeper or more regularized architectures. A visualization of this model's architecture can be seen in Appendix Fig. 2.

The motivation behind this model was very loosely based off of VGG16, which as we learned in class was a good starting point [1][2]. However, VGG has 5 blocks of convolution layers while ours only uses two. This is because the convolution layers in VGG16 architechture downsize the input image by a factor of 2. However, our input is quite small at 1x28x28, so we decided the two layers would be enough. Since this CNN is quite shallow, we used a kernel of size 5 instead to extract more features.

3.2 Enhanced CNN

To improve upon the baseline, we implemented an Enhanced CNN that incorporates techniques such as batch normalization and additional dropout layers. This model is deeper, using four convolutional layers grouped into two blocks. Each block consists of two convolutional layers with 3×3 filters, followed by batch normalization and ReLU activation. Max pooling and dropout (0.25) are applied after each block to reduce spatial dimensionality and prevent overfitting. After flattening, a fully connected layer with 512 units is used, followed by batch normalization and a dropout layer (0.5), before finally outputting class probabilities through a log-softmax function. The inclusion of batch normalization helps stabilize and accelerate training by reducing internal covariate shift, while dropout serves as a regularization technique to improve generalization performance. The architecture of this model is inspired by common practices seen in modern CNNs used in competitive image classification tasks. A visualization of this model's architecture can be seen in Appendix Fig. 3.

The two key changes here are batch normalization and more dropout [3]. Batch normalization improves training stability by keeping the input distributions to each layer consistent, even as earlier layers update during training. This helps the model train faster and more reliably. On the other hand, dropout randomly disables a fraction of neurons during training, forcing the network to learn more robust and redundant representations, which helps prevent overfitting.

3.3 ResNet

While we already faced the overfitting problem, another common issue faced when training deep neural networks is the degradation problem. As networks become deeper, their performance can begin to plateau or even worsen because the optimization becomes increasingly difficult. This is heightened by having such a small input image. To address this, we implemented a model inspired by the ResNet-18 architecture, which introduces residual connections to facilitate the training of very

deep networks. These connections act as shortcuts, allowing information and gradients to bypass one or more layers, effectively enabling the network to learn residual functions rather than trying to learn the entire transformation directly. This design eases the flow of gradients during backpropagation, improving convergence and enabling the successful training of much deeper models.

We modified the original ResNet architecture, which was designed for large RGB images, to suit the MNIST dataset of grayscale 28×28 digit images [4]. The model begins with a convolutional layer followed by batch normalization and ReLU activation, and proceeds through four stages of residual blocks with increasing filter sizes (64, 128, 256, 512). Downsampling and dimensionality matching are performed using 1×1 convolutions in the shortcut paths. After the final stage, global average pooling is applied to compress each feature map into a single value, followed by a fully connected layer for classification. A visualization of this model's architecture can be seen in Appendix Fig. 4. We experimented with several variations on the ResNet architecture, but ultimately found that the ResNet18-based model was the most accurate for this problem.

3.4 Hyperparameters

Through literature review, we determined that the best optimizer to use was Adam (Adaptive Moment Estimation)[5]. Initally, we trained each model many times to test a range of different learning rates, generally from 0.05-0.0001, to ensure we were getting the best possible performance. We tested training for different amounts of epochs, however we found that generally after 10-15 there is little to no improvement in performance, and beyond 20 the performance even decreases sometimes, likely due to overfitting.

Next, we went further by using AdamW, adding a weight decay for L2 regularization. We also added a learning rate scheduler, which dynamically adjusts the learning rate during training to improve model convergence [6]. Early in training, it allows larger steps to speed up learning and explore the loss landscape. As training progresses, it reduces the learning rate to fine-tune the model weights more precisely, helping avoid overshooting minima and settling into better optima.

4 Results

4.1 Model Comparisons

Due to the nature of our model choices, we expected performance to measurably increase as we progressed from the Baseline CNN to Enhanced CNN to a ResNet structure.

The baseline CNN achieved a best validation accuracy of **95.19%** with a learning rate of 0.001. When trained with a scheduler, performance improved to **96.28%**, demonstrating the benefits of dynamic learning rate adjustment even on a simpler architecture. The best baseline CNN submitted to the Kaggle competition scored a **0.95166**, which is pretty consistent with the validation score.

The enhanced CNN made a significant jump, reaching a peak **97.57%** test accuracy with the same learning rate, and improving even further to **98.38%** with the scheduler. These results indicate that architectural enhancements such as batch normalization and deeper layers contribute significantly to performance, especially when combined with effective training strategies. The best test accuracy here was a submission on **0.979**.

Finally, the ResNet model yielded the strongest results. With an optimal learning rate of 0.005, it achieved **97.76%** accuracy on the test set. When coupled with a learning rate scheduler, accuracy rose to **98.42%**. In this case, the Kaggle submission actually outperformed the validation accuracy, giving us our top score of **0.98833**.

4.2 Summary

Below is a table summarizing the validation accuracies at each stage of our investigations.

Method	Learning Rate	Validation Accuracy
Baseline CNN	fixed (0.001)	95.19%
Baseline CNN	scheduler	96.28%
Enhanced CNN	fixed (0.001)	97.57%
Enhanced CNN	scheduler	98.38%
ResNet	fixed (0.005)	97.76%
ResNet	scheduler	**98.42%**

Table 1: Summary of Validation Accuracy Results for Different Methods

5 Discussions and Conclusions

5.1 Discussion

Through our experiments, we observed a clear trend: deeper and more carefully regularized architectures consistently outperformed simpler baselines. Starting from a basic CNN inspired by VGG, we incrementally improved model performance by introducing batch normalization, dropout, and additional convolutional layers in our Enhanced CNN. These changes significantly increased both training stability and generalization performance.

The most notable gains came from the implementation of a ResNet-like architecture. Residual connections allowed us to build a much deeper network without suffering from vanishing gradients or degradation in accuracy. This proved especially valuable given the small input size of the MNIST images, which typically limits the effectiveness of deeper models. Additionally, using a learning rate scheduler improved the convergence speed and final accuracy of all models. Scheduling allowed for larger learning rates early in training, which helped the models escape shallow local minima, and smaller learning rates later, which helped fine-tune weights and reduce validation loss.

5.2 Future Improvements

Although our models performed very well, there are several potential avenues for improvement. One direction would be to explore data augmentation techniques such as rotation, scaling, or random erasing, which could help further reduce overfitting and improve robustness, especially for more complex datasets. Another area of exploration could be the use of more recent architectures such as EfficientNet or MobileNet, which provide excellent performance with fewer parameters and reduced computational cost. Additionally, ensembling multiple models could help increase prediction accuracy further, particularly on noisy or ambiguous inputs.

5.3 Conclusion

In conclusion, our study shows the significant impact that model architecture and training strategies have on image classification performance. Starting from a simple CNN, we achieved incremental improvements through architectural refinements and the use of learning rate scheduling, ending in a high-performing ResNet-based model. These findings underscore the importance of both model design and training methodology in achieving the best results. With the final ResNet + scheduler model reaching a Kaggle score of 0.98833, our approach demonstrates strong generalization and competitive accuracy on the MNIST dataset.

6 Statement of Contributions

The work was divided equally between each group member:

Yeo - Model research and experimentation, additional ResNet models

Kaitlyn - ResNet model, learning rate experimentation, writing report

Melody - Baseline and enhanced CNN models, learning rate scheduler experiments.

References

[1] Lecture Slides from ECSE-551: Machine Learning for Engineers by Prof. Armanfard

[2] B. Chintapalli, "VGG16: Convolutional neural network model," Built In, [Online]. Available: https://builtin.com/machine-learning/vgg16.

[3] A. Basli, "Regularization techniques in deep learning: Dropout, L-norm and batch normalization with examples," Medium, Jan. 30, 2020. [Online]. Available: https://medium.com/@adelbasli/regularization-techniques-in-deep-learning-dropout-l-norm-and-batch-normalization-with-3fe36bbbd353.

[4] "ResNet," Papers With Code, [Online]. Available: https://paperswithcode.com/method/resnet.

[5] V. Sharma, "A comprehensive guide on deep learning optimizers," Analytics Vidhya, Oct. 13, 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/.

[6] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Learning rate scheduling," in Dive into Deep Learning, [Online]. Available: https://d2l.ai/chapter_optimization/lr-scheduler.html.

Appendix A: Report Figures

The following section contains all figures referenced in the report. Some figures are also available in better quality in the attached .ipynb file.

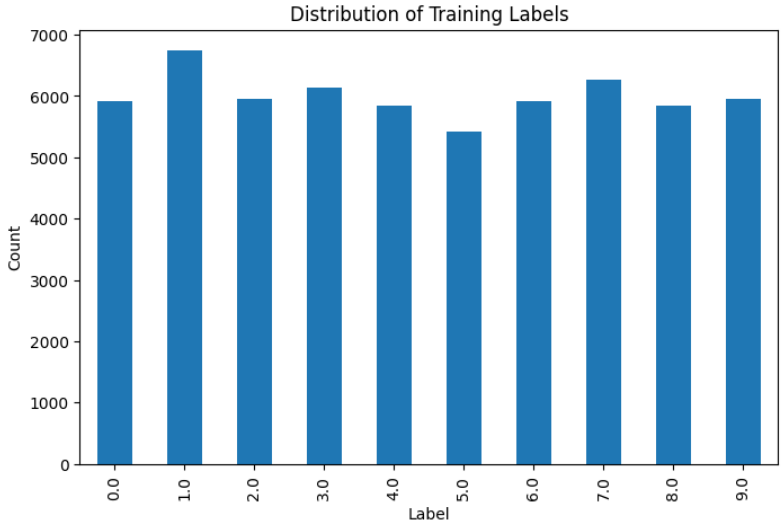


Figure 1: Class Distributions for the MNIST dataset

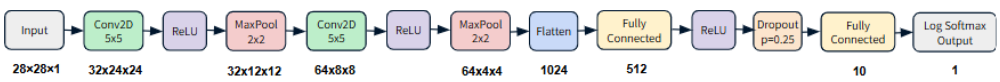


Figure 2: Architecture Model for Baseline CNN

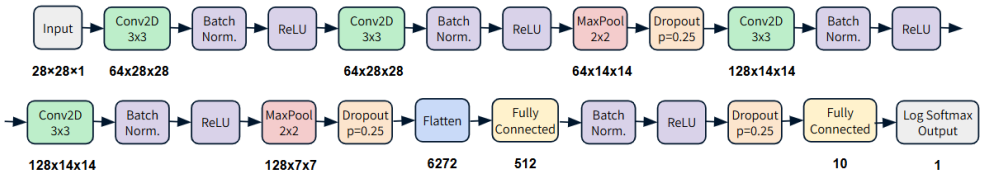


Figure 3: Architecture Model for Enhanced CNN

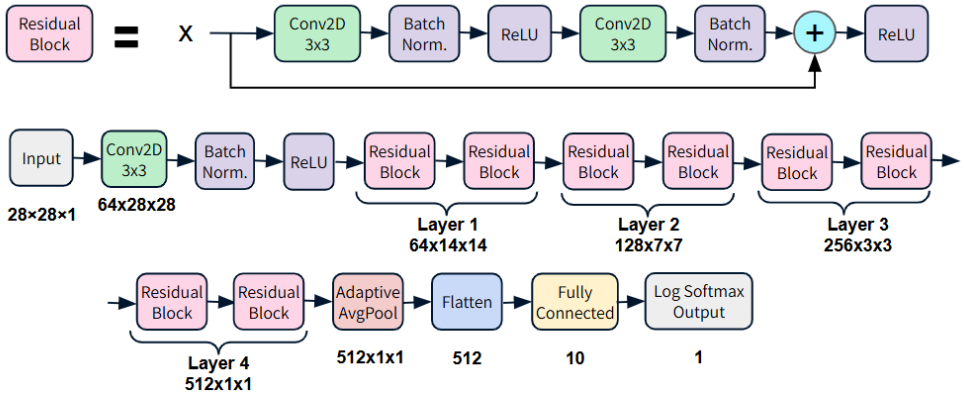


Figure 4: Architecture Model for ResNet

Appendix B: Code

The following section contains the code in the form of a pdf file of the Jupiter Notebook. We recommend opening the actual code file, submitted alongside this report, to view the full output blocks and all relevant code/analysis.

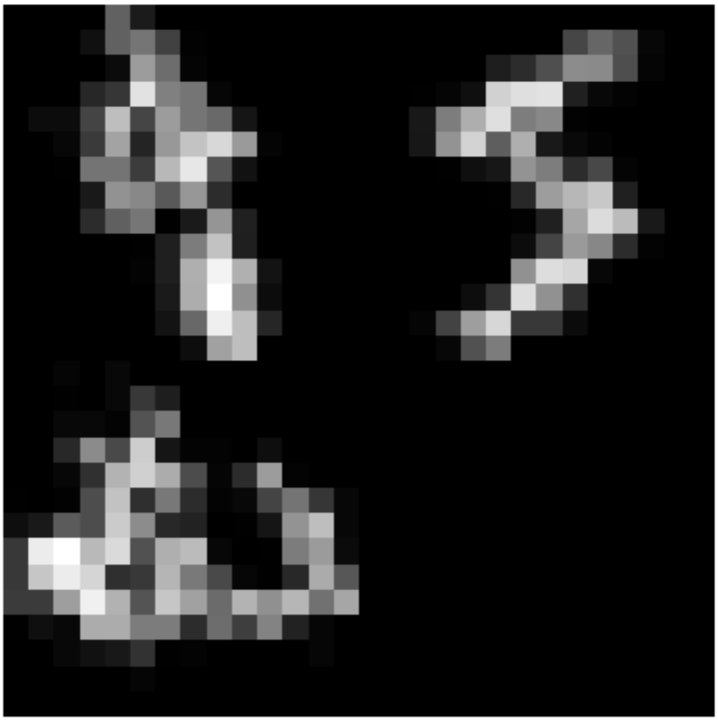
ECSE551Assignment3

April 11, 2025

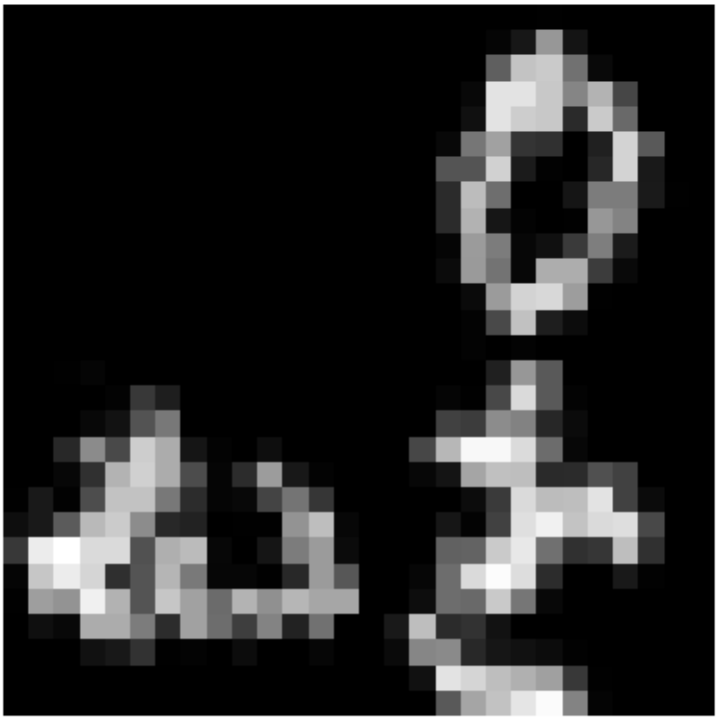
1 Data Exploration

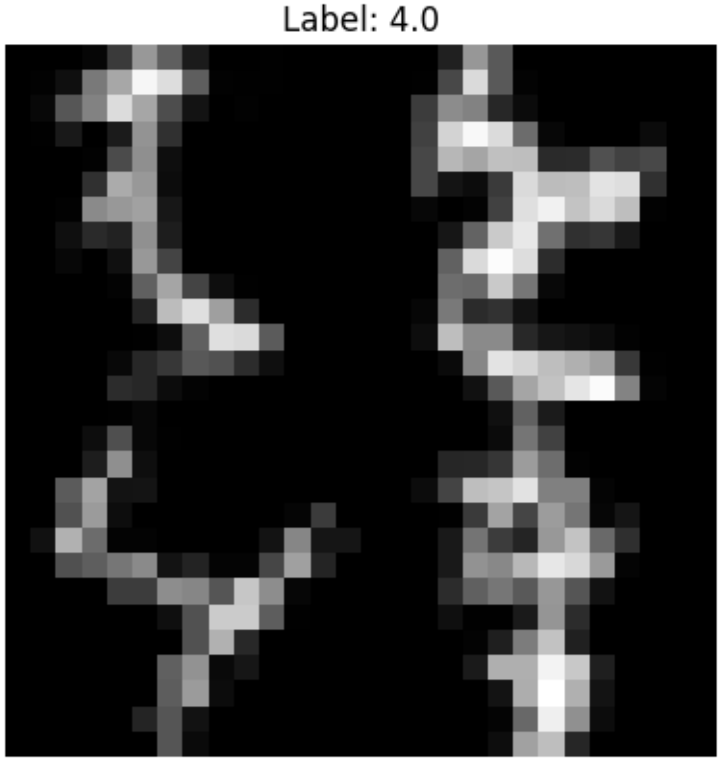
[ ]:
```
import pickle
import pandas as pd
import matplotlib.pyplot as plt

# Load labels from CSV
labels_df = pd.read_csv("Train_labels.csv", header=None)

# Load training data from pickle file
with open("Train.pkl", "rb") as f:
    train_data = pickle.load(f)

# Display first few images and their labels
for i in range(5):  # Display first 5 images
    plt.imshow(train_data[i].reshape((28, 28)), cmap='gray')  # Adjust cmap if␣
 ↪needed
    plt.title(f"Label: {labels_df.iloc[i, 0]}")
    plt.axis("off")
    plt.show()
```

Label: 5.0

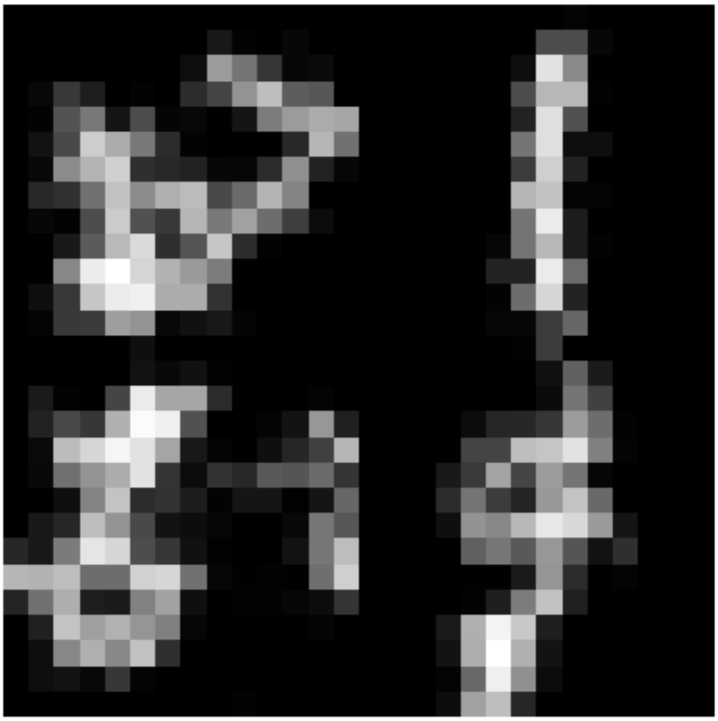


Label: 0.0

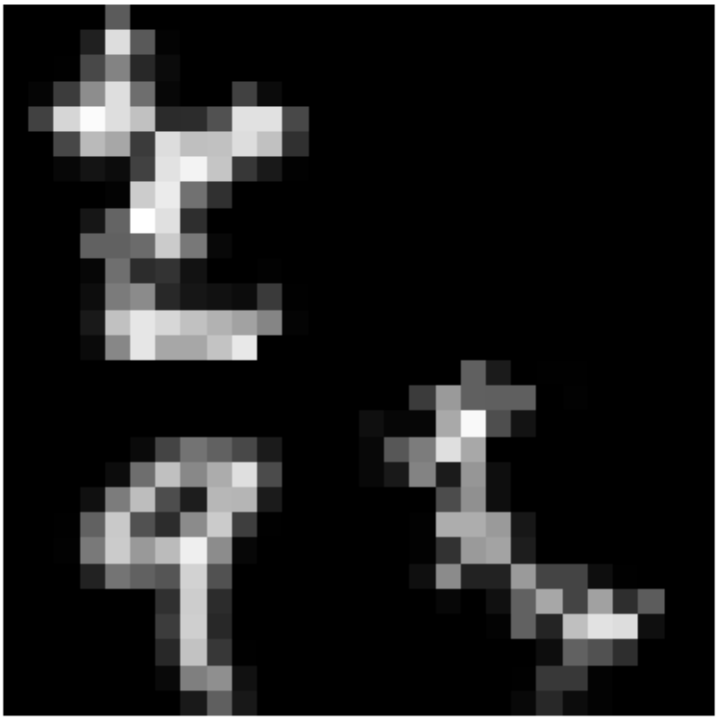



Label: 4.0

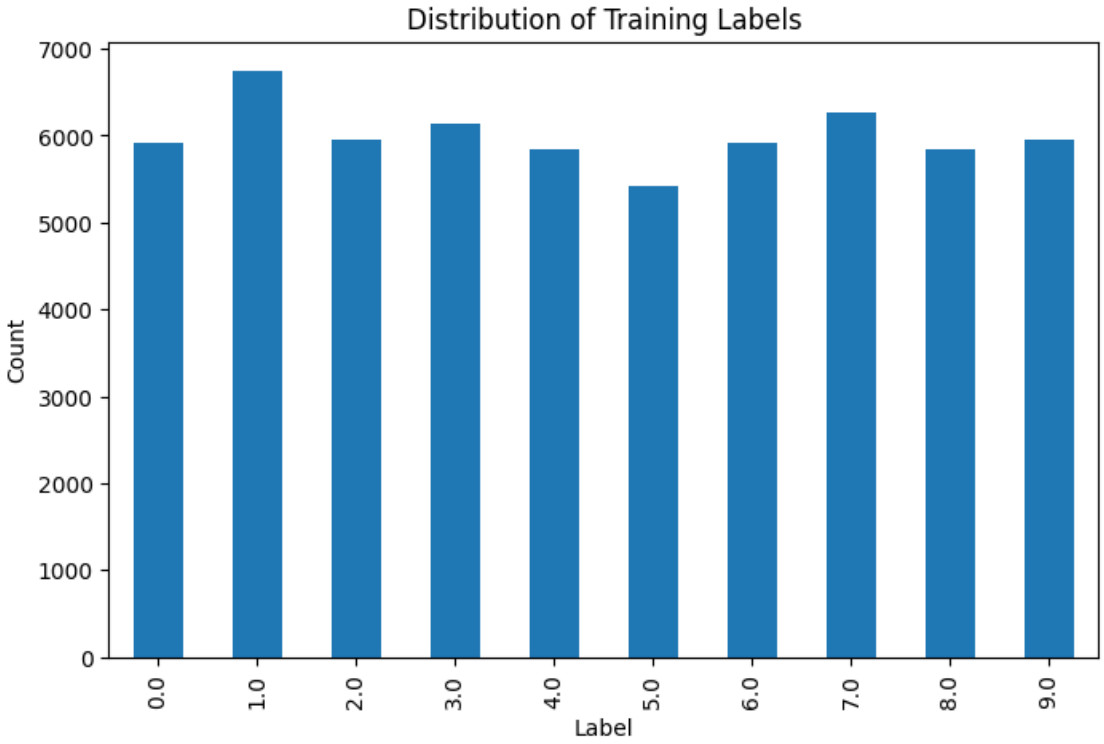
Label: 1.0



Label: 9.0



[ ]:
```
plt.figure(figsize=(8, 5))
labels_df[0].value_counts().sort_index().plot(kind='bar')
plt.xlabel("Label")
plt.ylabel("Count")
plt.title("Distribution of Training Labels")
plt.show()
print(labels_df[0].value_counts())
```




```
0
1.0    6742
7.0    6265
3.0    6131
2.0    5958
9.0    5949
0.0    5923
6.0    5918
8.0    5851
4.0    5842
5.0    5421
Name: count, dtype: int64
```

2 Model 1: Baseline CNN

Define Model

[ ]:
```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class CNN(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()
        # First convolutional layer
        # Input: 1x28x28, Output: 32x24x24
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=0)
        # First pooling layer
        # Input: 32x24x24, Output: 32x12x12
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Second convolutional layer
        # Input: 32x12x12, Output: 64x8x8
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=0)
        # Second pooling layer
        # Input: 64x8x8, Output: 64x4x4
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, num_classes)

        # Dropout for regularization
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # First conv block
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        # Second conv block
        x = F.relu(self.conv2(x))
        x = self.pool2(x)

        # Flatten
        x = x.view(-1, 64 * 4 * 4)

        # Fully connected layers with ReLU and dropout
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
```

```
        return F.log_softmax(x, dim=1)
```

Normal Training

[ ]:
```
import torch
import torch.utils.data as data
import pandas as pd
import pickle
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader, random_split
import torch.nn as nn
import torch.optim as optim

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)


epochs = 10
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.long()

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Calculate epoch accuracy and loss
    epoch_accuracy = 100 * correct_train / total_train
    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
 ↪4f}, Accuracy: {epoch_accuracy:.2f}%")

# Save trained model
torch.save(model.state_dict(), "cnn_model.pth")


model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
```

```
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

```
Epoch 1/10, Loss: 1.4840, Accuracy: 46.80%
Epoch 2/10, Loss: 0.7207, Accuracy: 75.70%
Epoch 3/10, Loss: 0.4990, Accuracy: 83.95%
Epoch 4/10, Loss: 0.3793, Accuracy: 88.01%
Epoch 5/10, Loss: 0.3209, Accuracy: 89.85%
Epoch 6/10, Loss: 0.2800, Accuracy: 91.34%
Epoch 7/10, Loss: 0.2533, Accuracy: 92.10%
Epoch 8/10, Loss: 0.2317, Accuracy: 92.79%
Epoch 9/10, Loss: 0.2144, Accuracy: 93.28%
Epoch 10/10, Loss: 0.2025, Accuracy: 93.61%
Test Accuracy: 93.06%
```

Train with different learning rates

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
```

```
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Train Models with Different LRs
learning_rates = [0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]
results = {}

for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    model = CNN(num_classes=10)
    model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    train_accuracies = []
    val_accuracies = []

    epochs = 10
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            labels = labels.long()

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

```
            running_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        # Calculate epoch accuracy and loss
        epoch_accuracy = 100 * correct_train / total_train
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):
↪.4f}, Accuracy: {epoch_accuracy:.2f}%")

        results[lr] = {
            'train_accuracies': train_accuracies,
            'val_accuracies': val_accuracies
        }
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f"Final Test Accuracy: {100 * correct / total:.2f}%")

    torch.save(model.state_dict(), f"cnn_lr_{lr}.pth")
```

```
Training with learning rate: 0.05
Epoch 1/10, Loss: 2.5798, Accuracy: 10.65%
Epoch 2/10, Loss: 2.3057, Accuracy: 10.44%
Epoch 3/10, Loss: 2.3047, Accuracy: 10.60%
Epoch 4/10, Loss: 2.3054, Accuracy: 10.71%
Epoch 5/10, Loss: 2.3059, Accuracy: 10.37%
Epoch 6/10, Loss: 2.3057, Accuracy: 10.64%
Epoch 7/10, Loss: 2.3050, Accuracy: 10.56%
Epoch 8/10, Loss: 2.3054, Accuracy: 10.61%
Epoch 9/10, Loss: 2.3053, Accuracy: 10.54%
Epoch 10/10, Loss: 2.3053, Accuracy: 10.60%
Final Test Accuracy: 9.79%
Training with learning rate: 0.01
Epoch 1/10, Loss: 2.3030, Accuracy: 11.04%
Epoch 2/10, Loss: 2.3019, Accuracy: 11.15%
Epoch 3/10, Loss: 2.3020, Accuracy: 11.19%
Epoch 4/10, Loss: 2.3020, Accuracy: 11.20%
```

```
Epoch 5/10, Loss: 2.3021, Accuracy: 11.16%
Epoch 6/10, Loss: 2.3021, Accuracy: 11.07%
Epoch 7/10, Loss: 2.3019, Accuracy: 11.16%
Epoch 8/10, Loss: 2.3020, Accuracy: 11.12%
Epoch 9/10, Loss: 2.3021, Accuracy: 11.14%
Epoch 10/10, Loss: 2.3020, Accuracy: 11.21%
Final Test Accuracy: 11.00%
Training with learning rate: 0.005
Epoch 1/10, Loss: 2.3024, Accuracy: 11.09%
Epoch 2/10, Loss: 2.3016, Accuracy: 11.30%
Epoch 3/10, Loss: 2.3016, Accuracy: 11.22%
Epoch 4/10, Loss: 2.3014, Accuracy: 11.27%
Epoch 5/10, Loss: 2.3016, Accuracy: 11.24%
Epoch 6/10, Loss: 2.3016, Accuracy: 11.30%
Epoch 7/10, Loss: 2.3015, Accuracy: 11.30%
Epoch 8/10, Loss: 2.3016, Accuracy: 11.18%
Epoch 9/10, Loss: 2.3015, Accuracy: 11.30%
Epoch 10/10, Loss: 2.3016, Accuracy: 11.30%
Final Test Accuracy: 11.00%
Training with learning rate: 0.001
Epoch 1/10, Loss: 1.2463, Accuracy: 55.29%
Epoch 2/10, Loss: 0.4342, Accuracy: 86.14%
Epoch 3/10, Loss: 0.2813, Accuracy: 91.39%
Epoch 4/10, Loss: 0.2207, Accuracy: 93.28%
Epoch 5/10, Loss: 0.1825, Accuracy: 94.30%
Epoch 6/10, Loss: 0.1554, Accuracy: 95.11%
Epoch 7/10, Loss: 0.1405, Accuracy: 95.58%
Epoch 8/10, Loss: 0.1238, Accuracy: 96.07%
Epoch 9/10, Loss: 0.1115, Accuracy: 96.43%
Epoch 10/10, Loss: 0.1012, Accuracy: 96.72%
Final Test Accuracy: 95.19%
Training with learning rate: 0.0005
Epoch 1/10, Loss: 1.4656, Accuracy: 47.92%
Epoch 2/10, Loss: 0.6113, Accuracy: 80.00%
Epoch 3/10, Loss: 0.4087, Accuracy: 86.90%
Epoch 4/10, Loss: 0.3148, Accuracy: 90.05%
Epoch 5/10, Loss: 0.2596, Accuracy: 91.89%
Epoch 6/10, Loss: 0.2210, Accuracy: 92.99%
Epoch 7/10, Loss: 0.1913, Accuracy: 93.98%
Epoch 8/10, Loss: 0.1671, Accuracy: 94.69%
Epoch 9/10, Loss: 0.1486, Accuracy: 95.20%
Epoch 10/10, Loss: 0.1313, Accuracy: 95.82%
Final Test Accuracy: 94.29%
Training with learning rate: 0.0001
Epoch 1/10, Loss: 2.1270, Accuracy: 23.69%
Epoch 2/10, Loss: 1.3471, Accuracy: 55.18%
Epoch 3/10, Loss: 1.0083, Accuracy: 67.01%
Epoch 4/10, Loss: 0.8392, Accuracy: 72.92%
```

```
Epoch 5/10, Loss: 0.7299, Accuracy: 76.42%
Epoch 6/10, Loss: 0.6525, Accuracy: 79.09%
Epoch 7/10, Loss: 0.5918, Accuracy: 81.04%
Epoch 8/10, Loss: 0.5434, Accuracy: 82.70%
Epoch 9/10, Loss: 0.5031, Accuracy: 84.04%
Epoch 10/10, Loss: 0.4678, Accuracy: 85.28%
Final Test Accuracy: 86.47%
```

Optimized training with LR scheduler

[ ]:
```python
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])
```

```
# Increased batch size for faster training
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Initialize the EnhancedCNN model
model = CNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()

# Set to 15 epochs
epochs = 15

# Using AdamW optimizer with weight decay
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-5)

# Learning rate scheduler for faster convergence - adjusted for 15 epochs
scheduler = optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=0.01,
    steps_per_epoch=len(train_loader),
    epochs=epochs
)

# Training loop with 15 epochs
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.long()

        optimizer.zero_grad()
        outputs = model(images)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()  # Step the scheduler

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
```

```
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Calculate epoch accuracy and loss
    epoch_accuracy = 100 * correct_train / total_train
    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
 ↪4f}, Accuracy: {epoch_accuracy:.2f}%, LR: {scheduler.get_last_lr()[0]:.6f}")

    # Optional: Evaluate on test set after each 5 epochs
    if (epoch + 1) % 5 == 0:
        model.eval()
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                test_total += labels.size(0)
                test_correct += (predicted == labels).sum().item()

        test_accuracy = 100 * test_correct / test_total
        print(f"Test Accuracy at Epoch {epoch+1}: {test_accuracy:.2f}%")
        model.train()  # Switch back to training mode

# Save trained model
torch.save(model.state_dict(), "enhanced_cnn_model_15epochs.pth")

# Final Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Final Test Accuracy: {100 * correct / total:.2f}%")
```

```
Using device: cuda
Epoch 1/15, Loss: 1.7728, Accuracy: 36.29%, LR: 0.001524
Epoch 2/15, Loss: 0.6869, Accuracy: 77.01%, LR: 0.004370
Epoch 3/15, Loss: 0.3117, Accuracy: 90.44%, LR: 0.007605
Epoch 4/15, Loss: 0.2263, Accuracy: 93.35%, LR: 0.009713
```

```
Epoch 5/15, Loss: 0.1965, Accuracy: 94.39%, LR: 0.009944
Test Accuracy at Epoch 5: 93.14%
Epoch 6/15, Loss: 0.1889, Accuracy: 94.53%, LR: 0.009503
Epoch 7/15, Loss: 0.1580, Accuracy: 95.48%, LR: 0.008663
Epoch 8/15, Loss: 0.1350, Accuracy: 96.12%, LR: 0.007497
Epoch 9/15, Loss: 0.1171, Accuracy: 96.69%, LR: 0.006109
Epoch 10/15, Loss: 0.0942, Accuracy: 97.11%, LR: 0.004622
Test Accuracy at Epoch 10: 95.94%
Epoch 11/15, Loss: 0.0683, Accuracy: 98.00%, LR: 0.003170
Epoch 12/15, Loss: 0.0483, Accuracy: 98.46%, LR: 0.001879
Epoch 13/15, Loss: 0.0335, Accuracy: 98.94%, LR: 0.000867
Epoch 14/15, Loss: 0.0240, Accuracy: 99.26%, LR: 0.000221
Epoch 15/15, Loss: 0.0203, Accuracy: 99.40%, LR: 0.000000
Test Accuracy at Epoch 15: 96.28%
Final Test Accuracy: 96.28%
```

Save Predictions

[ ]:
```python
# Load test data
with open("Test.pkl", "rb") as f:
    test_images = pickle.load(f)


class ImageDataset(Dataset):
    def __init__(self, images):
        self.images = images

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        image = image.astype('float32')  # Ensure image is float32 for␣
 ↪normalization
        return image


test_dataset = ImageDataset(test_images)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.load_state_dict(torch.load("enhanced_cnn_model_15epochs.pth"))
model.eval()  # Set the model to evaluation mode

# Make predictions on the test dataset
predictions = []
ids = []

with torch.no_grad():
```

```
    for idx, images in enumerate(test_loader):
        images = images.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        batch_size = images.size(0)
        start_id = idx * 64 + 1  # IDs start from 1
        for i in range(batch_size):
            ids.append(start_id + i)
            predictions.append(predicted[i].item())

# Create a DataFrame with 'id' and 'label' columns
df = pd.DataFrame({
    'id': ids,
    'class': predictions
})

# Save the predictions to a CSV file
df.to_csv('predictions1.csv', index=False)
print("Predictions saved to 'predictions1.csv'")
```

```
<ipython-input-41-29815f405276>:22: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model.load_state_dict(torch.load("enhanced_cnn_model_15epochs.pth"))

Predictions saved to 'predictions1.csv'
```

3 Model 2: Optimized CNN

Define Model

[ ]:
```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class EnhancedCNN(nn.Module):
```

```
    def __init__(self, num_classes=10):
        super(EnhancedCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(2)
        self.dropout1 = nn.Dropout(0.25)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(2)
        self.dropout2 = nn.Dropout(0.25)

        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 7 * 7, 512)
        self.bn5 = nn.BatchNorm1d(512)
        self.dropout3 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool1(x)
        x = self.dropout1(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.bn4(self.conv4(x)))
        x = self.pool2(x)
        x = self.dropout2(x)

        x = self.flatten(x)
        x = F.relu(self.bn5(self.fc1(x)))
        x = self.dropout3(x)
        x = self.fc2(x)

        return F.log_softmax(x, dim=1)
```

Normal training

[ ]:
```
import torch
import torch.utils.data as data
import pandas as pd
import pickle
import torchvision.transforms as transforms
```

```
from torch.utils.data import Dataset, DataLoader, random_split
import torch.nn as nn
import torch.optim as optim

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = EnhancedCNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 10
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
```

```
    total_train = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.long()

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Calculate epoch accuracy and loss
    epoch_accuracy = 100 * correct_train / total_train
    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
 ↪4f}, Accuracy: {epoch_accuracy:.2f}%")

# Save trained model
torch.save(model.state_dict(), "cnn_model.pth")


model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

```
Epoch 1/10, Loss: 0.5972, Accuracy: 80.35%
Epoch 2/10, Loss: 0.2269, Accuracy: 92.78%
Epoch 3/10, Loss: 0.1708, Accuracy: 94.52%
Epoch 4/10, Loss: 0.1386, Accuracy: 95.63%
Epoch 5/10, Loss: 0.1186, Accuracy: 96.25%
Epoch 6/10, Loss: 0.1060, Accuracy: 96.64%
Epoch 7/10, Loss: 0.0922, Accuracy: 96.98%
```

```
Epoch 8/10, Loss: 0.0832, Accuracy: 97.28%
Epoch 9/10, Loss: 0.0702, Accuracy: 97.69%
Epoch 10/10, Loss: 0.0669, Accuracy: 97.77%
Test Accuracy: 97.37%
```

Train with different learning rates

[ ]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])
```

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Train Models with Different LRs
learning_rates = [0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]
results = {}

for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    model = EnhancedCNN(num_classes=10)
    model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    train_accuracies = []
    val_accuracies = []

    epochs = 10
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            labels = labels.long()

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        # Calculate epoch accuracy and loss
        epoch_accuracy = 100 * correct_train / total_train
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):
↪.4f}, Accuracy: {epoch_accuracy:.2f}%")
```

```
        results[lr] = {
            'train_accuracies': train_accuracies,
            'val_accuracies': val_accuracies
        }
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f"Final Test Accuracy: {100 * correct / total:.2f}%")

    torch.save(model.state_dict(), f"cnn_lr_{lr}.pth")
```

```
Training with learning rate: 0.05
Epoch 1/10, Loss: 1.1454, Accuracy: 63.19%
Epoch 2/10, Loss: 0.4616, Accuracy: 86.46%
Epoch 3/10, Loss: 0.3759, Accuracy: 89.14%
Epoch 4/10, Loss: 0.3312, Accuracy: 90.66%
Epoch 5/10, Loss: 0.2970, Accuracy: 91.57%
Epoch 6/10, Loss: 0.2862, Accuracy: 91.99%
Epoch 7/10, Loss: 0.2634, Accuracy: 92.72%
Epoch 8/10, Loss: 0.2511, Accuracy: 93.00%
Epoch 9/10, Loss: 0.2429, Accuracy: 93.34%
Epoch 10/10, Loss: 0.2272, Accuracy: 93.68%
Final Test Accuracy: 96.42%
Training with learning rate: 0.01
Epoch 1/10, Loss: 0.6294, Accuracy: 79.44%
Epoch 2/10, Loss: 0.2360, Accuracy: 92.91%
Epoch 3/10, Loss: 0.1852, Accuracy: 94.39%
Epoch 4/10, Loss: 0.1635, Accuracy: 95.11%
Epoch 5/10, Loss: 0.1475, Accuracy: 95.58%
Epoch 6/10, Loss: 0.1309, Accuracy: 96.12%
Epoch 7/10, Loss: 0.1253, Accuracy: 96.29%
Epoch 8/10, Loss: 0.1206, Accuracy: 96.36%
Epoch 9/10, Loss: 0.1075, Accuracy: 96.79%
Epoch 10/10, Loss: 0.1025, Accuracy: 96.97%
Final Test Accuracy: 97.20%
Training with learning rate: 0.005
Epoch 1/10, Loss: 0.5855, Accuracy: 80.56%
Epoch 2/10, Loss: 0.2202, Accuracy: 93.06%
Epoch 3/10, Loss: 0.1638, Accuracy: 94.95%
```

```
Epoch 4/10, Loss: 0.1434, Accuracy: 95.57%
Epoch 5/10, Loss: 0.1229, Accuracy: 96.29%
Epoch 6/10, Loss: 0.1117, Accuracy: 96.53%
Epoch 7/10, Loss: 0.1028, Accuracy: 96.81%
Epoch 8/10, Loss: 0.0941, Accuracy: 97.04%
Epoch 9/10, Loss: 0.0832, Accuracy: 97.37%
Epoch 10/10, Loss: 0.0808, Accuracy: 97.50%
Final Test Accuracy: 97.30%
Training with learning rate: 0.001
Epoch 1/10, Loss: 0.5855, Accuracy: 80.79%
Epoch 2/10, Loss: 0.2216, Accuracy: 92.96%
Epoch 3/10, Loss: 0.1731, Accuracy: 94.55%
Epoch 4/10, Loss: 0.1418, Accuracy: 95.43%
Epoch 5/10, Loss: 0.1231, Accuracy: 96.11%
Epoch 6/10, Loss: 0.1051, Accuracy: 96.51%
Epoch 7/10, Loss: 0.0937, Accuracy: 96.89%
Epoch 8/10, Loss: 0.0810, Accuracy: 97.38%
Epoch 9/10, Loss: 0.0738, Accuracy: 97.61%
Epoch 10/10, Loss: 0.0662, Accuracy: 97.82%
Final Test Accuracy: 97.57%
Training with learning rate: 0.0005
Epoch 1/10, Loss: 0.6612, Accuracy: 78.38%
Epoch 2/10, Loss: 0.2423, Accuracy: 92.28%
Epoch 3/10, Loss: 0.1821, Accuracy: 94.16%
Epoch 4/10, Loss: 0.1545, Accuracy: 95.15%
Epoch 5/10, Loss: 0.1305, Accuracy: 95.91%
Epoch 6/10, Loss: 0.1152, Accuracy: 96.34%
Epoch 7/10, Loss: 0.1028, Accuracy: 96.66%
Epoch 8/10, Loss: 0.0911, Accuracy: 97.02%
Epoch 9/10, Loss: 0.0810, Accuracy: 97.25%
Epoch 10/10, Loss: 0.0729, Accuracy: 97.65%
Final Test Accuracy: 97.49%
Training with learning rate: 0.0001
Epoch 1/10, Loss: 1.1711, Accuracy: 61.88%
Epoch 2/10, Loss: 0.4110, Accuracy: 87.62%
Epoch 3/10, Loss: 0.2881, Accuracy: 91.13%
Epoch 4/10, Loss: 0.2342, Accuracy: 92.70%
Epoch 5/10, Loss: 0.2021, Accuracy: 93.59%
Epoch 6/10, Loss: 0.1728, Accuracy: 94.56%
Epoch 7/10, Loss: 0.1537, Accuracy: 95.11%
Epoch 8/10, Loss: 0.1428, Accuracy: 95.38%
Epoch 9/10, Loss: 0.1323, Accuracy: 95.73%
Epoch 10/10, Loss: 0.1184, Accuracy: 96.21%
Final Test Accuracy: 96.45%
```

Optimized training parameters for EnhancedCNN

[ ]:
```
# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

# Increased batch size for faster training
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Initialize the EnhancedCNN model
model = EnhancedCNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()

# Using AdamW optimizer with weight decay
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-4)

# Learning rate scheduler for faster convergence
scheduler = optim.lr_scheduler.OneCycleLR(
    optimizer,
```

```
    max_lr=0.01,
    steps_per_epoch=len(train_loader),
    epochs=10
)

# Training loop
epochs = 10
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.long()

        optimizer.zero_grad()
        outputs = model(images)

        # The model outputs log_softmax, so we need to use NLLLoss instead of␣
 ↪CrossEntropyLoss
        loss = F.nll_loss(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()  # Step the scheduler

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Calculate epoch accuracy and loss
    epoch_accuracy = 100 * correct_train / total_train
    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
 ↪4f}, Accuracy: {epoch_accuracy:.2f}%, LR: {scheduler.get_last_lr()[0]:.6f}")

# Save trained model
torch.save(model.state_dict(), "enhanced_cnn_model.pth")

# Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
```

```
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

```
Using device: cuda
Epoch 1/10, Loss: 0.7994, Accuracy: 73.52%, LR: 0.002804
Epoch 2/10, Loss: 0.3014, Accuracy: 90.51%, LR: 0.007608
Epoch 3/10, Loss: 0.2191, Accuracy: 93.29%, LR: 0.010000
Epoch 4/10, Loss: 0.1628, Accuracy: 95.06%, LR: 0.009502
Epoch 5/10, Loss: 0.1297, Accuracy: 95.98%, LR: 0.008113
Epoch 6/10, Loss: 0.0988, Accuracy: 96.93%, LR: 0.006107
Epoch 7/10, Loss: 0.0697, Accuracy: 97.67%, LR: 0.003882
Epoch 8/10, Loss: 0.0473, Accuracy: 98.49%, LR: 0.001878
Epoch 9/10, Loss: 0.0320, Accuracy: 98.92%, LR: 0.000493
Epoch 10/10, Loss: 0.0228, Accuracy: 99.27%, LR: 0.000000
Test Accuracy: 98.03%
```

Optimized training with LR scheduler

[ ]:
```
# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)
```

```
train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

# Increased batch size for faster training
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Initialize the EnhancedCNN model
model = EnhancedCNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()

# Set to 15 epochs
epochs = 15

# Using AdamW optimizer with weight decay
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-4)

# Learning rate scheduler for faster convergence - adjusted for 15 epochs
scheduler = optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=0.01,
    steps_per_epoch=len(train_loader),
    epochs=epochs
)

# Training loop with 15 epochs
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.long()

        optimizer.zero_grad()
        outputs = model(images)

        # The model outputs log_softmax, so we need to use NLLLoss
        loss = F.nll_loss(outputs, labels)
        loss.backward()
```

```
        optimizer.step()
        scheduler.step()  # Step the scheduler

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Calculate epoch accuracy and loss
    epoch_accuracy = 100 * correct_train / total_train
    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
 ↪4f}, Accuracy: {epoch_accuracy:.2f}%, LR: {scheduler.get_last_lr()[0]:.6f}")

    # Optional: Evaluate on test set after each 5 epochs
    if (epoch + 1) % 5 == 0:
        model.eval()
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                test_total += labels.size(0)
                test_correct += (predicted == labels).sum().item()

        test_accuracy = 100 * test_correct / test_total
        print(f"Test Accuracy at Epoch {epoch+1}: {test_accuracy:.2f}%")
        model.train()  # Switch back to training mode

# Save trained model
torch.save(model.state_dict(), "enhanced_cnn_model_15epochs.pth")

# Final Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Final Test Accuracy: {100 * correct / total:.2f}%")
```

```
Using device: cuda
Epoch 1/15, Loss: 0.8162, Accuracy: 73.10%, LR: 0.001524
Epoch 2/15, Loss: 0.3018, Accuracy: 90.48%, LR: 0.004370
Epoch 3/15, Loss: 0.2373, Accuracy: 92.55%, LR: 0.007605
Epoch 4/15, Loss: 0.1902, Accuracy: 94.13%, LR: 0.009713
Epoch 5/15, Loss: 0.1564, Accuracy: 95.27%, LR: 0.009944
Test Accuracy at Epoch 5: 96.30%
Epoch 6/15, Loss: 0.1312, Accuracy: 95.87%, LR: 0.009503
Epoch 7/15, Loss: 0.1107, Accuracy: 96.62%, LR: 0.008663
Epoch 8/15, Loss: 0.0883, Accuracy: 97.25%, LR: 0.007497
Epoch 9/15, Loss: 0.0757, Accuracy: 97.54%, LR: 0.006109
Epoch 10/15, Loss: 0.0611, Accuracy: 98.08%, LR: 0.004622
Test Accuracy at Epoch 10: 97.75%
Epoch 11/15, Loss: 0.0430, Accuracy: 98.57%, LR: 0.003170
Epoch 12/15, Loss: 0.0300, Accuracy: 99.00%, LR: 0.001879
Epoch 13/15, Loss: 0.0201, Accuracy: 99.35%, LR: 0.000867
Epoch 14/15, Loss: 0.0147, Accuracy: 99.53%, LR: 0.000221
Epoch 15/15, Loss: 0.0120, Accuracy: 99.63%, LR: 0.000000
Test Accuracy at Epoch 15: 98.38%
Final Test Accuracy: 98.38%
```

Save Predictions

[ ]:
```python
# Load test data
with open("Test.pkl", "rb") as f:
    test_images = pickle.load(f)


class ImageDataset(Dataset):
    def __init__(self, images):
        self.images = images

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        image = image.astype('float32')  # Ensure image is float32 for␣
 ↪normalization
        return image


test_dataset = ImageDataset(test_images)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.load_state_dict(torch.load("enhanced_cnn_model_15epochs.pth"))
model.eval()  # Set the model to evaluation mode
```

```
# Make predictions on the test dataset
predictions = []
ids = []

with torch.no_grad():
    for idx, images in enumerate(test_loader):
        images = images.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        batch_size = images.size(0)
        start_id = idx * 64 + 1  # IDs start from 1
        for i in range(batch_size):
            ids.append(start_id + i)
            predictions.append(predicted[i].item())

# Create a DataFrame with 'id' and 'label' columns
df = pd.DataFrame({
    'id': ids,
    'class': predictions
})

# Save the predictions to a CSV file
df.to_csv('predictions1.csv', index=False)
print("Predictions saved to 'predictions1.csv'")
```

```
<ipython-input-41-29815f405276>:22: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model.load_state_dict(torch.load("enhanced_cnn_model_15epochs.pth"))

Predictions saved to 'predictions1.csv'
```

[ ]:
```
# Load test data
with open("Test.pkl", "rb") as f:
    test_images = pickle.load(f)
```

```
class ImageDataset(Dataset):
    def __init__(self, images):
        self.images = images

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        image = image.astype('float32')  # Ensure image is float32 for␣
 ↪normalization
        return image


test_dataset = ImageDataset(test_images)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.load_state_dict(torch.load("cnn_model.pth"))
model.eval()  # Set the model to evaluation mode

# Make predictions on the test dataset
predictions = []
ids = []

with torch.no_grad():
    for idx, images in enumerate(test_loader):
        images = images.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        batch_size = images.size(0)
        start_id = idx * 64 + 1  # IDs start from 1
        for i in range(batch_size):
            ids.append(start_id + i)
            predictions.append(predicted[i].item())

# Create a DataFrame with 'id' and 'label' columns
df = pd.DataFrame({
    'id': ids,
    'class': predictions
})

# Save the predictions to a CSV file
df.to_csv('predictions.csv', index=False)
print("Predictions saved to 'predictions.csv'")
```

```
Predictions saved to 'predictions.csv'
```

4 Model 3: ResNet

Define Model

[ ]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        nn.Module.__init__(self)
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
 ↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
 ↪stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        identity = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = F.relu(out)
        return out


class ResNet18CNN(nn.Module):
    def __init__(self, num_classes=10):
        nn.Module.__init__(self)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1,␣
 ↪bias=False)
        self.bn1 = nn.BatchNorm2d(64)

        self.layer1 = self._make_layer(64, 64, 2, stride=1)
        self.layer2 = self._make_layer(64, 128, 2, stride=2)
        self.layer3 = self._make_layer(128, 256, 2, stride=2)
        self.layer4 = self._make_layer(256, 512, 2, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)
```

```
    def _make_layer(self, in_channels, out_channels, blocks, stride):
        downsample = None
        if stride != 1 or in_channels != out_channels:
            downsample = nn.Sequential(nn.Conv2d(in_channels, out_channels,␣
↪kernel_size=1,stride=stride, bias=False), nn.BatchNorm2d(out_channels))

        layers = []
        layers.append(ResidualBlock(in_channels, out_channels, stride,␣
↪downsample))
        for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return F.log_softmax(x, dim=1)
```

Train with different learning rates

[ ]:
```
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
```

```
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Train Models with Different LRs
learning_rates = [0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]
results = {}

for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    model = ResNet18CNN(num_classes=10)
    model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    train_accuracies = []
    val_accuracies = []

    epochs = 10
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0
```

```
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            labels = labels.long()

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        # Calculate epoch accuracy and loss
        epoch_accuracy = 100 * correct_train / total_train
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):
↪.4f}, Accuracy: {epoch_accuracy:.2f}%")

        results[lr] = {
            'train_accuracies': train_accuracies,
            'val_accuracies': val_accuracies
        }
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f"Final Test Accuracy: {100 * correct / total:.2f}%")

    torch.save(model.state_dict(), f"cnn_lr_{lr}.pth")
```

```
Training with learning rate: 0.05
Epoch 1/10, Loss: 1.3047, Accuracy: 55.49%
Epoch 2/10, Loss: 0.2199, Accuracy: 93.24%
Epoch 3/10, Loss: 0.1618, Accuracy: 95.00%
Epoch 4/10, Loss: 0.1315, Accuracy: 96.00%
Epoch 5/10, Loss: 0.1138, Accuracy: 96.56%
Epoch 6/10, Loss: 0.0985, Accuracy: 97.03%
```

```
Epoch 7/10, Loss: 0.0905, Accuracy: 97.21%
Epoch 8/10, Loss: 0.0814, Accuracy: 97.52%
Epoch 9/10, Loss: 0.0736, Accuracy: 97.72%
Epoch 10/10, Loss: 0.0659, Accuracy: 97.95%
Final Test Accuracy: 96.18%
Training with learning rate: 0.01
Epoch 1/10, Loss: 0.8906, Accuracy: 69.49%
Epoch 2/10, Loss: 0.1841, Accuracy: 94.49%
Epoch 3/10, Loss: 0.1322, Accuracy: 96.01%
Epoch 4/10, Loss: 0.1085, Accuracy: 96.70%
Epoch 5/10, Loss: 0.0887, Accuracy: 97.25%
Epoch 6/10, Loss: 0.0779, Accuracy: 97.56%
Epoch 7/10, Loss: 0.0660, Accuracy: 97.86%
Epoch 8/10, Loss: 0.0553, Accuracy: 98.20%
Epoch 9/10, Loss: 0.0509, Accuracy: 98.36%
Epoch 10/10, Loss: 0.0444, Accuracy: 98.60%
Final Test Accuracy: 96.51%
Training with learning rate: 0.005
Epoch 1/10, Loss: 0.5571, Accuracy: 81.42%
Epoch 2/10, Loss: 0.1614, Accuracy: 95.10%
Epoch 3/10, Loss: 0.1162, Accuracy: 96.52%
Epoch 4/10, Loss: 0.0943, Accuracy: 97.21%
Epoch 5/10, Loss: 0.0782, Accuracy: 97.66%
Epoch 6/10, Loss: 0.0662, Accuracy: 98.01%
Epoch 7/10, Loss: 0.0542, Accuracy: 98.29%
Epoch 8/10, Loss: 0.0488, Accuracy: 98.52%
Epoch 9/10, Loss: 0.0381, Accuracy: 98.81%
Epoch 10/10, Loss: 0.0367, Accuracy: 98.85%
Final Test Accuracy: 97.76%
Training with learning rate: 0.001
Epoch 1/10, Loss: 0.3875, Accuracy: 87.36%
Epoch 2/10, Loss: 0.1467, Accuracy: 95.51%
Epoch 3/10, Loss: 0.1085, Accuracy: 96.67%
Epoch 4/10, Loss: 0.0865, Accuracy: 97.35%
Epoch 5/10, Loss: 0.0760, Accuracy: 97.65%
Epoch 6/10, Loss: 0.0611, Accuracy: 98.06%
Epoch 7/10, Loss: 0.0490, Accuracy: 98.41%
Epoch 8/10, Loss: 0.0456, Accuracy: 98.53%
Epoch 9/10, Loss: 0.0388, Accuracy: 98.78%
Epoch 10/10, Loss: 0.0329, Accuracy: 98.94%
Final Test Accuracy: 96.28%
Training with learning rate: 0.0005
Epoch 1/10, Loss: 0.3744, Accuracy: 87.75%
Epoch 2/10, Loss: 0.1395, Accuracy: 95.67%
Epoch 3/10, Loss: 0.1025, Accuracy: 96.92%
Epoch 4/10, Loss: 0.0840, Accuracy: 97.37%
Epoch 5/10, Loss: 0.0677, Accuracy: 97.81%
Epoch 6/10, Loss: 0.0557, Accuracy: 98.21%
```

```
Epoch 7/10, Loss: 0.0466, Accuracy: 98.51%
Epoch 8/10, Loss: 0.0426, Accuracy: 98.62%
Epoch 9/10, Loss: 0.0365, Accuracy: 98.82%
Epoch 10/10, Loss: 0.0296, Accuracy: 99.05%
Final Test Accuracy: 96.82%
Training with learning rate: 0.0001
Epoch 1/10, Loss: 0.4900, Accuracy: 83.84%
Epoch 2/10, Loss: 0.1377, Accuracy: 95.60%
Epoch 3/10, Loss: 0.0895, Accuracy: 97.22%
Epoch 4/10, Loss: 0.0640, Accuracy: 98.05%
Epoch 5/10, Loss: 0.0510, Accuracy: 98.35%
Epoch 6/10, Loss: 0.0411, Accuracy: 98.62%
Epoch 7/10, Loss: 0.0375, Accuracy: 98.74%
Epoch 8/10, Loss: 0.0288, Accuracy: 99.01%
Epoch 9/10, Loss: 0.0297, Accuracy: 99.01%
Epoch 10/10, Loss: 0.0217, Accuracy: 99.25%
Final Test Accuracy: 95.91%
```

Optimized training with LR scheduler

[ ]:
```python
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
```

```
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

# Increased batch size for faster training
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Initialize the EnhancedCNN model
model = ResNet18CNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()

# Set to 15 epochs
epochs = 15

# Using AdamW optimizer with weight decay
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-5)

# Learning rate scheduler for faster convergence - adjusted for 15 epochs
scheduler = optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=0.01,
    steps_per_epoch=len(train_loader),
    epochs=epochs
)

# Training loop with 15 epochs
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.long()

        optimizer.zero_grad()
```

```
        outputs = model(images)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()  # Step the scheduler

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Calculate epoch accuracy and loss
    epoch_accuracy = 100 * correct_train / total_train
    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
 ↪4f}, Accuracy: {epoch_accuracy:.2f}%, LR: {scheduler.get_last_lr()[0]:.6f}")

    # Optional: Evaluate on test set after each 5 epochs
    if (epoch + 1) % 5 == 0:
        model.eval()
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                test_total += labels.size(0)
                test_correct += (predicted == labels).sum().item()

        test_accuracy = 100 * test_correct / test_total
        print(f"Test Accuracy at Epoch {epoch+1}: {test_accuracy:.2f}%")
        model.train()  # Switch back to training mode

# Save trained model
torch.save(model.state_dict(), "enhanced_cnn_model_15epochs.pth")

# Final Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
```

```
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Final Test Accuracy: {100 * correct / total:.2f}%")
```

```
Using device: cuda
Epoch 1/15, Loss: 0.4761, Accuracy: 84.28%, LR: 0.001524
Epoch 2/15, Loss: 0.1983, Accuracy: 94.06%, LR: 0.004370
Epoch 3/15, Loss: 0.1643, Accuracy: 95.16%, LR: 0.007605
Epoch 4/15, Loss: 0.1300, Accuracy: 96.17%, LR: 0.009713
Epoch 5/15, Loss: 0.0960, Accuracy: 97.15%, LR: 0.009944
Test Accuracy at Epoch 5: 96.21%
Epoch 6/15, Loss: 0.0717, Accuracy: 97.89%, LR: 0.009503
Epoch 7/15, Loss: 0.0597, Accuracy: 98.17%, LR: 0.008663
Epoch 8/15, Loss: 0.0434, Accuracy: 98.65%, LR: 0.007497
Epoch 9/15, Loss: 0.0322, Accuracy: 99.00%, LR: 0.006109
Epoch 10/15, Loss: 0.0204, Accuracy: 99.30%, LR: 0.004622
Test Accuracy at Epoch 10: 97.55%
Epoch 11/15, Loss: 0.0105, Accuracy: 99.67%, LR: 0.003170
Epoch 12/15, Loss: 0.0042, Accuracy: 99.88%, LR: 0.001879
Epoch 13/15, Loss: 0.0006, Accuracy: 100.00%, LR: 0.000867
Epoch 14/15, Loss: 0.0003, Accuracy: 100.00%, LR: 0.000221
Epoch 15/15, Loss: 0.0002, Accuracy: 100.00%, LR: 0.000000
Test Accuracy at Epoch 15: 98.42%
Final Test Accuracy: 98.42%
```

Save Predictions

[ ]:
```
# Load test data
with open("Test.pkl", "rb") as f:
    test_images = pickle.load(f)


class ImageDataset(Dataset):
    def __init__(self, images):
        self.images = images

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        image = image.astype('float32')  # Ensure image is float32 for␣
 ↪normalization
        return image


test_dataset = ImageDataset(test_images)
```

```
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.load_state_dict(torch.load("enhanced_cnn_model_15epochs.pth"))
model.eval()  # Set the model to evaluation mode

# Make predictions on the test dataset
predictions = []
ids = []

with torch.no_grad():
    for idx, images in enumerate(test_loader):
        images = images.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        batch_size = images.size(0)
        start_id = idx * 64 + 1  # IDs start from 1
        for i in range(batch_size):
            ids.append(start_id + i)
            predictions.append(predicted[i].item())

# Create a DataFrame with 'id' and 'label' columns
df = pd.DataFrame({
    'id': ids,
    'class': predictions
})

# Save the predictions to a CSV file
df.to_csv('predictions.csv', index=False)
print("Predictions saved to 'predictions.csv'")
```

```
Predictions saved to 'predictions.csv'
```

5 Other ResNet Experiments

We also tried ResNet34 and ResNet9.

ResNet34

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
```

```
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        identity = x
        if self.downsample:
            identity = self.downsample(x)

        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += identity
        return self.relu(out)
import torch
import torch.nn as nn
import torch.nn.functional as F

class ResNet34CNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1,␣
 ↪bias=False)  # QMNIST is grayscale
        self.bn1 = nn.BatchNorm2d(64)

        self.layer1 = self._make_layer(64, 64, blocks=3, stride=1)
        self.layer2 = self._make_layer(64, 128, blocks=4, stride=2)
        self.layer3 = self._make_layer(128, 256, blocks=6, stride=2)
        self.layer4 = self._make_layer(256, 512, blocks=3, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, in_channels, out_channels, blocks, stride):
        downsample = None
        if stride != 1 or in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,␣
 ↪stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

        layers = [ResidualBlock(in_channels, out_channels, stride, downsample)]
        for _ in range(1, blocks):
```

```
            layers.append(ResidualBlock(out_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return F.log_softmax(x, dim=1)
```

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]
```

```
        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
# Train Models with Different LRs
learning_rates = [0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]
results = {}


for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    model = ResNet34CNN(num_classes=10)
    model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    #optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=1e-5)
    criterion = nn.CrossEntropyLoss()

    train_accuracies = []
    val_accuracies = []

    epochs = 10
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            labels = labels.long()

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
```

```
            optimizer.step()

            running_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        # Calculate epoch accuracy and loss
        epoch_accuracy = 100 * correct_train / total_train
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):
↪.4f}, Accuracy: {epoch_accuracy:.2f}%")

        results[lr] = {
            'train_accuracies': train_accuracies,
            'val_accuracies': val_accuracies
        }
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f"Final Test Accuracy: {100 * correct / total:.2f}%")

    torch.save(model.state_dict(), f"cnn_lr_{lr}.pth")
```

```
cuda
Training with learning rate: 0.05
Epoch 1/10, Loss: 1.6974, Accuracy: 41.50%
Epoch 2/10, Loss: 0.2703, Accuracy: 91.84%
Epoch 3/10, Loss: 0.1714, Accuracy: 94.76%
Epoch 4/10, Loss: 0.1429, Accuracy: 95.77%
Epoch 5/10, Loss: 0.1205, Accuracy: 96.39%
Epoch 6/10, Loss: 0.1074, Accuracy: 96.78%
Epoch 7/10, Loss: 0.1019, Accuracy: 96.98%
Epoch 8/10, Loss: 0.0961, Accuracy: 97.15%
Epoch 9/10, Loss: 0.0849, Accuracy: 97.45%
Epoch 10/10, Loss: 0.0776, Accuracy: 97.68%
Final Test Accuracy: 96.19%
Training with learning rate: 0.01
Epoch 1/10, Loss: 1.7436, Accuracy: 36.52%
```

```
Epoch 2/10, Loss: 0.2662, Accuracy: 91.91%
Epoch 3/10, Loss: 0.1584, Accuracy: 95.15%
Epoch 4/10, Loss: 0.1226, Accuracy: 96.21%
Epoch 5/10, Loss: 0.1010, Accuracy: 96.92%
Epoch 6/10, Loss: 0.0858, Accuracy: 97.39%
Epoch 7/10, Loss: 0.0798, Accuracy: 97.55%
Epoch 8/10, Loss: 0.0663, Accuracy: 97.90%
Epoch 9/10, Loss: 0.0624, Accuracy: 98.08%
Epoch 10/10, Loss: 0.0519, Accuracy: 98.38%
Final Test Accuracy: 97.50%
Training with learning rate: 0.005
Epoch 1/10, Loss: 1.4642, Accuracy: 47.37%
Epoch 2/10, Loss: 0.2295, Accuracy: 93.06%
Epoch 3/10, Loss: 0.1528, Accuracy: 95.36%
Epoch 4/10, Loss: 0.1160, Accuracy: 96.43%
Epoch 5/10, Loss: 0.0970, Accuracy: 96.99%
Epoch 6/10, Loss: 0.0827, Accuracy: 97.43%
Epoch 7/10, Loss: 0.0722, Accuracy: 97.74%
Epoch 8/10, Loss: 0.0605, Accuracy: 98.09%
Epoch 9/10, Loss: 0.0557, Accuracy: 98.27%
Epoch 10/10, Loss: 0.0452, Accuracy: 98.57%
Final Test Accuracy: 97.24%
Training with learning rate: 0.001
Epoch 1/10, Loss: 0.5006, Accuracy: 83.36%
Epoch 2/10, Loss: 0.1678, Accuracy: 95.02%
Epoch 3/10, Loss: 0.1238, Accuracy: 96.34%
Epoch 4/10, Loss: 0.1035, Accuracy: 96.89%
Epoch 5/10, Loss: 0.0874, Accuracy: 97.40%
Epoch 6/10, Loss: 0.0732, Accuracy: 97.77%
Epoch 7/10, Loss: 0.0622, Accuracy: 98.13%
Epoch 8/10, Loss: 0.0567, Accuracy: 98.16%
Epoch 9/10, Loss: 0.0487, Accuracy: 98.50%
Epoch 10/10, Loss: 0.0420, Accuracy: 98.68%
Final Test Accuracy: 97.19%
Training with learning rate: 0.0005
Epoch 1/10, Loss: 0.4875, Accuracy: 83.94%
Epoch 2/10, Loss: 0.1593, Accuracy: 95.26%
Epoch 3/10, Loss: 0.1180, Accuracy: 96.59%
Epoch 4/10, Loss: 0.0986, Accuracy: 96.94%
Epoch 5/10, Loss: 0.0800, Accuracy: 97.60%
Epoch 6/10, Loss: 0.0699, Accuracy: 97.89%
Epoch 7/10, Loss: 0.0574, Accuracy: 98.21%
Epoch 8/10, Loss: 0.0490, Accuracy: 98.49%
Epoch 9/10, Loss: 0.0489, Accuracy: 98.49%
Epoch 10/10, Loss: 0.0361, Accuracy: 98.85%
Final Test Accuracy: 97.33%
Training with learning rate: 0.0001
Epoch 1/10, Loss: 0.5824, Accuracy: 80.21%
```

```
Epoch 2/10, Loss: 0.1539, Accuracy: 95.17%
Epoch 3/10, Loss: 0.1033, Accuracy: 96.89%
Epoch 4/10, Loss: 0.0799, Accuracy: 97.52%
Epoch 5/10, Loss: 0.0646, Accuracy: 98.03%
Epoch 6/10, Loss: 0.0589, Accuracy: 98.21%
Epoch 7/10, Loss: 0.0462, Accuracy: 98.58%
Epoch 8/10, Loss: 0.0405, Accuracy: 98.77%
Epoch 9/10, Loss: 0.0397, Accuracy: 98.69%
Epoch 10/10, Loss: 0.0275, Accuracy: 99.15%
Final Test Accuracy: 97.07%
```

[ ]:
```python
# Load test data
with open("Test.pkl", "rb") as f:
    test_images = pickle.load(f)


class ImageDataset(Dataset):
    def __init__(self, images):
        self.images = images

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        image = image.astype('float32')  # Ensure image is float32 for␣
 ↪normalization
        return image


test_dataset = ImageDataset(test_images)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.load_state_dict(torch.load("cnn_lr_0.001.pth"))
model.eval()  # Set the model to evaluation mode

# Make predictions on the test dataset
predictions = []
ids = []

with torch.no_grad():
    for idx, images in enumerate(test_loader):
        images = images.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
```

```
        batch_size = images.size(0)
        start_id = idx * 64 + 1  # IDs start from 1
        for i in range(batch_size):
            ids.append(start_id + i)
            predictions.append(predicted[i].item())

# Create a DataFrame with 'id' and 'label' columns
df = pd.DataFrame({
    'id': ids,
    'class': predictions
})

# Save the predictions to a CSV file
df.to_csv('predictions.csv', index=False)
print("Predictions saved to 'predictions.csv'")
```

```
Predictions saved to 'predictions.csv'
```

ResNet34 yields 0.97966 accuracy on test set.

ResNet9

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        identity = x
        if self.downsample:
            identity = self.downsample(x)

        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += identity
        return self.relu(out)
```

```
class ResNet9(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()

        # Initial convolution layers
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1, bias=False), ␣
 ↪# QMNIST: grayscale
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )

        # Two residual blocks
        self.res1 = ResidualBlock(128, 128)
        self.conv3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        self.res2 = ResidualBlock(256, 256)

        self.pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.res1(x)
        x = self.conv3(x)
        x = self.res2(x)
        x = self.pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)
```

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
import numpy as np
```

```
import pandas as pd
import pickle
from sklearn.model_selection import StratifiedShuffleSplit

# Load training data
with open("Train.pkl", "rb") as f:
    train_images = pickle.load(f)

train_labels = pd.read_csv("Train_labels.csv", header=None)[0].values

class ImageDataset(Dataset):
    def __init__(self, images, labels):
        self.images = images
        self.labels = labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        image = image.astype('float32')
        image = image.reshape(1, 28, 28)
        label = torch.tensor(label, dtype=torch.long)
        return image, label

full_dataset = ImageDataset(train_images, train_labels)

train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size,␣
 ↪test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
# Train Models with Different LRs
learning_rates = [0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]
results = {}


for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    model = ResNet9(num_classes=10)
    model.to(device)
```

```
    optimizer = optim.Adam(model.parameters(), lr=lr,weight_decay=1e-5)
    criterion = nn.CrossEntropyLoss()

    train_accuracies = []
    val_accuracies = []

    epochs = 10
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            labels = labels.long()

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        # Calculate epoch accuracy and loss
        epoch_accuracy = 100 * correct_train / total_train
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):
↪.4f}, Accuracy: {epoch_accuracy:.2f}%")

        results[lr] = {
            'train_accuracies': train_accuracies,
            'val_accuracies': val_accuracies
        }
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
```

```
            correct += (predicted == labels).sum().item()

    print(f"Final Test Accuracy: {100 * correct / total:.2f}%")

    torch.save(model.state_dict(), f"cnn_lr_{lr}.pth")
```

```
cuda
Training with learning rate: 0.05
Epoch 1/10, Loss: 1.0342, Accuracy: 63.95%
Epoch 2/10, Loss: 0.2244, Accuracy: 93.19%
Epoch 3/10, Loss: 0.1822, Accuracy: 94.28%
Epoch 4/10, Loss: 0.1687, Accuracy: 94.78%
Epoch 5/10, Loss: 0.1609, Accuracy: 95.13%
Epoch 6/10, Loss: 0.1535, Accuracy: 95.28%
Epoch 7/10, Loss: 0.1534, Accuracy: 95.32%
Epoch 8/10, Loss: 0.1446, Accuracy: 95.55%
Epoch 9/10, Loss: 0.1456, Accuracy: 95.57%
Epoch 10/10, Loss: 0.1397, Accuracy: 95.70%
Final Test Accuracy: 94.17%
Training with learning rate: 0.01
Epoch 1/10, Loss: 0.6652, Accuracy: 76.89%
Epoch 2/10, Loss: 0.1567, Accuracy: 95.20%
Epoch 3/10, Loss: 0.1338, Accuracy: 95.83%
Epoch 4/10, Loss: 0.1110, Accuracy: 96.65%
Epoch 5/10, Loss: 0.1051, Accuracy: 96.75%
Epoch 6/10, Loss: 0.0936, Accuracy: 97.05%
Epoch 7/10, Loss: 0.0876, Accuracy: 97.27%
Epoch 8/10, Loss: 0.0826, Accuracy: 97.40%
Epoch 9/10, Loss: 0.0755, Accuracy: 97.60%
Epoch 10/10, Loss: 0.0717, Accuracy: 97.80%
Final Test Accuracy: 96.79%
Training with learning rate: 0.005
Epoch 1/10, Loss: 0.5951, Accuracy: 79.73%
Epoch 2/10, Loss: 0.1484, Accuracy: 95.42%
Epoch 3/10, Loss: 0.1232, Accuracy: 96.20%
Epoch 4/10, Loss: 0.1057, Accuracy: 96.70%
Epoch 5/10, Loss: 0.0955, Accuracy: 97.03%
Epoch 6/10, Loss: 0.0835, Accuracy: 97.40%
Epoch 7/10, Loss: 0.0787, Accuracy: 97.49%
Epoch 8/10, Loss: 0.0729, Accuracy: 97.70%
Epoch 9/10, Loss: 0.0644, Accuracy: 97.93%
Epoch 10/10, Loss: 0.0587, Accuracy: 98.14%
Final Test Accuracy: 97.67%
Training with learning rate: 0.001
Epoch 1/10, Loss: 0.4097, Accuracy: 88.32%
Epoch 2/10, Loss: 0.1312, Accuracy: 96.00%
Epoch 3/10, Loss: 0.0970, Accuracy: 97.06%
```

```
Epoch 4/10, Loss: 0.0829, Accuracy: 97.45%
Epoch 5/10, Loss: 0.0684, Accuracy: 97.79%
Epoch 6/10, Loss: 0.0598, Accuracy: 98.07%
Epoch 7/10, Loss: 0.0503, Accuracy: 98.36%
Epoch 8/10, Loss: 0.0414, Accuracy: 98.64%
Epoch 9/10, Loss: 0.0389, Accuracy: 98.66%
Epoch 10/10, Loss: 0.0352, Accuracy: 98.83%
Final Test Accuracy: 97.26%
Training with learning rate: 0.0005
Epoch 1/10, Loss: 0.4146, Accuracy: 88.99%
Epoch 2/10, Loss: 0.1294, Accuracy: 96.20%
Epoch 3/10, Loss: 0.0955, Accuracy: 97.05%
Epoch 4/10, Loss: 0.0781, Accuracy: 97.51%
Epoch 5/10, Loss: 0.0646, Accuracy: 97.96%
Epoch 6/10, Loss: 0.0546, Accuracy: 98.29%
Epoch 7/10, Loss: 0.0472, Accuracy: 98.52%
Epoch 8/10, Loss: 0.0394, Accuracy: 98.71%
Epoch 9/10, Loss: 0.0343, Accuracy: 98.87%
Epoch 10/10, Loss: 0.0248, Accuracy: 99.20%
Final Test Accuracy: 96.68%
Training with learning rate: 0.0001
Epoch 1/10, Loss: 0.7110, Accuracy: 83.91%
Epoch 2/10, Loss: 0.1873, Accuracy: 95.94%
Epoch 3/10, Loss: 0.1231, Accuracy: 97.01%
Epoch 4/10, Loss: 0.0906, Accuracy: 97.70%
Epoch 5/10, Loss: 0.0711, Accuracy: 98.11%
Epoch 6/10, Loss: 0.0555, Accuracy: 98.55%
Epoch 7/10, Loss: 0.0457, Accuracy: 98.76%
Epoch 8/10, Loss: 0.0348, Accuracy: 99.10%
Epoch 9/10, Loss: 0.0275, Accuracy: 99.31%
Epoch 10/10, Loss: 0.0277, Accuracy: 99.27%
Final Test Accuracy: 93.66%
```

[ ]:
```python
# Load test data
with open("Test.pkl", "rb") as f:
    test_images = pickle.load(f)


class ImageDataset(Dataset):
    def __init__(self, images):
        self.images = images

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
```

```
        image = image.astype('float32')  # Ensure image is float32 for␣
 ↪normalization
        return image


test_dataset = ImageDataset(test_images)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.load_state_dict(torch.load("cnn_lr_0.005.pth"))
model.eval()  # Set the model to evaluation mode

# Make predictions on the test dataset
predictions = []
ids = []

with torch.no_grad():
    for idx, images in enumerate(test_loader):
        images = images.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        batch_size = images.size(0)
        start_id = idx * 64 + 1  # IDs start from 1
        for i in range(batch_size):
            ids.append(start_id + i)
            predictions.append(predicted[i].item())

# Create a DataFrame with 'id' and 'label' columns
df = pd.DataFrame({
    'id': ids,
    'class': predictions
})

# Save the predictions to a CSV file
df.to_csv('predictions.csv', index=False)
print("Predictions saved to 'predictions.csv'")
```

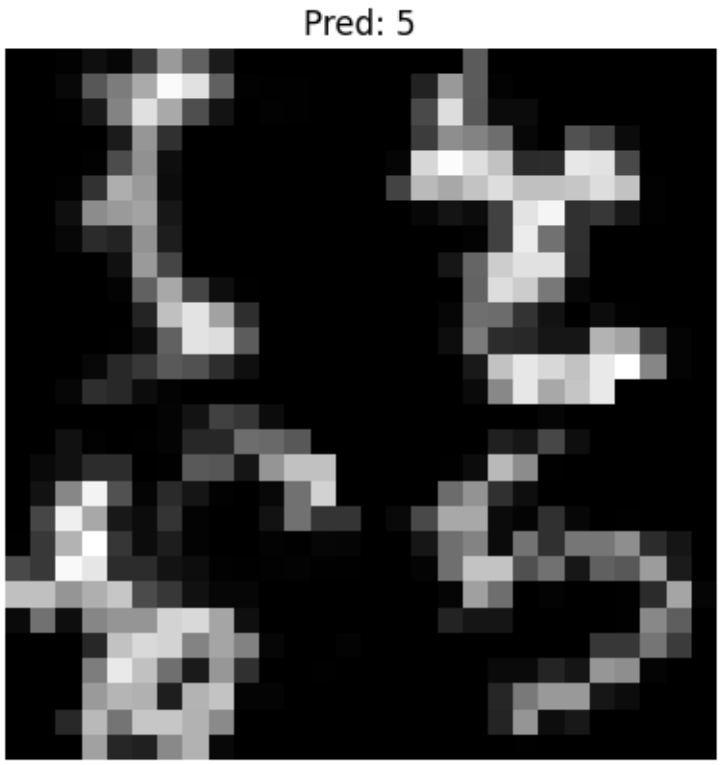
```
Predictions saved to 'predictions.csv'
```

ResNet9 is not preforming better than ResNet34 and ResNet18

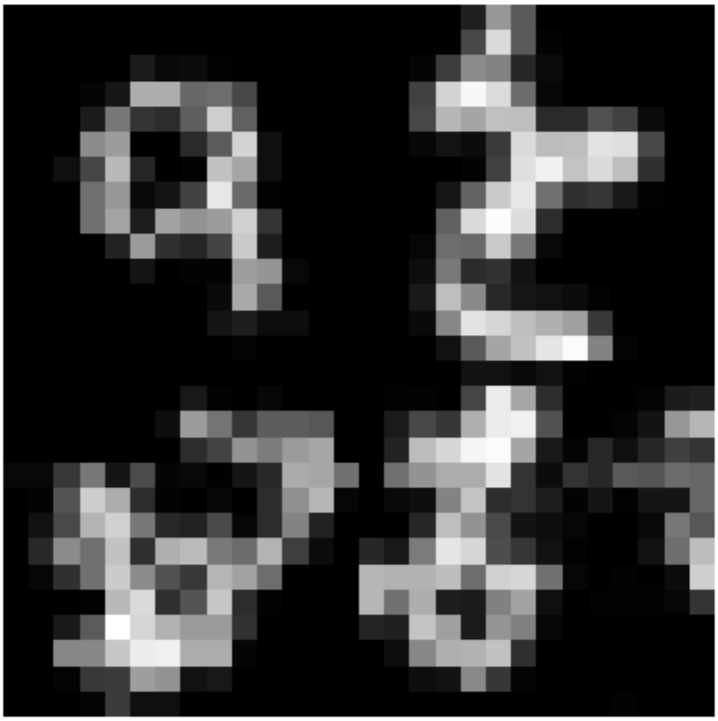
Show a few sample predictions

[ ]:
```
import matplotlib.pyplot as plt

# Display first few images and their labels
for i in range(10):
```

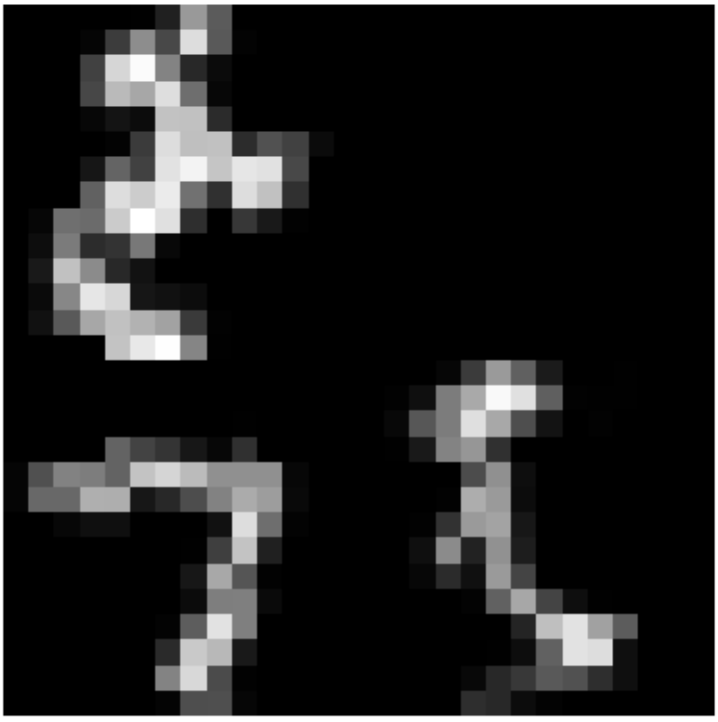
```
    plt.imshow(test_images[i+15].reshape((28, 28)), cmap='gray')
    plt.title(f"Pred: {predictions[i+15]}")
    plt.axis("off")
    plt.show()
```

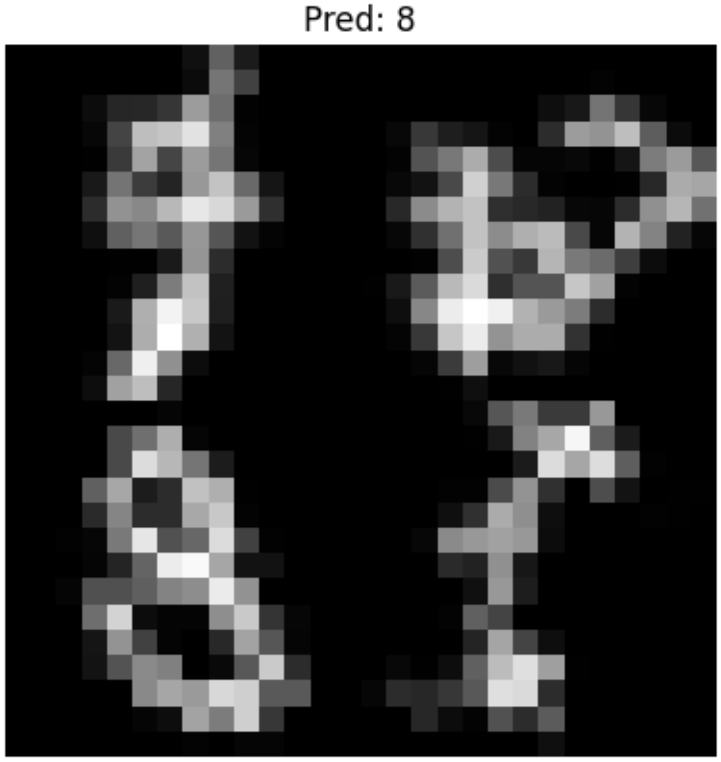


Pred: 9

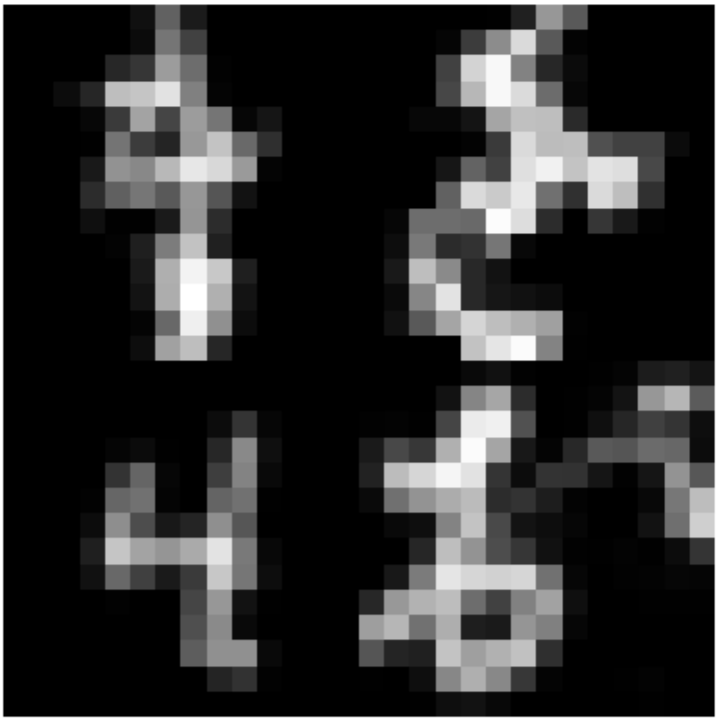


Pred: 7

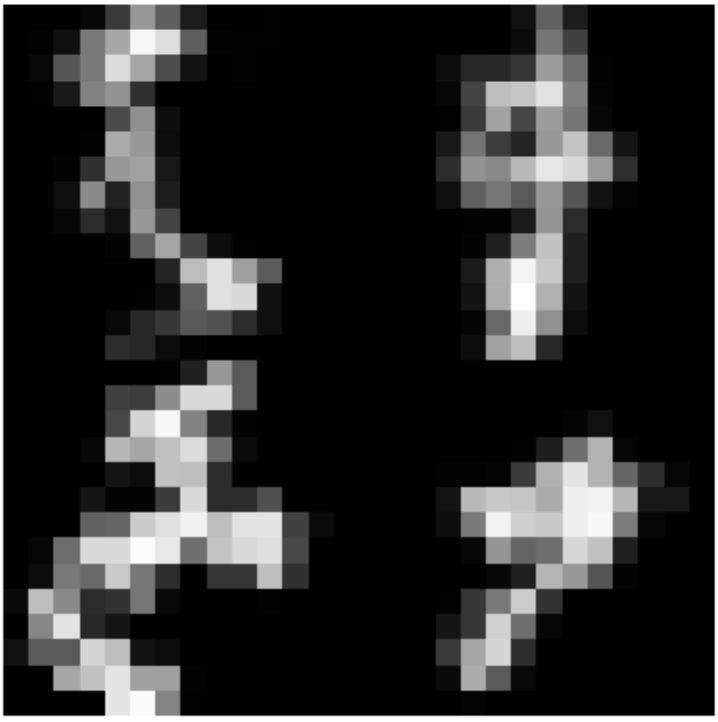


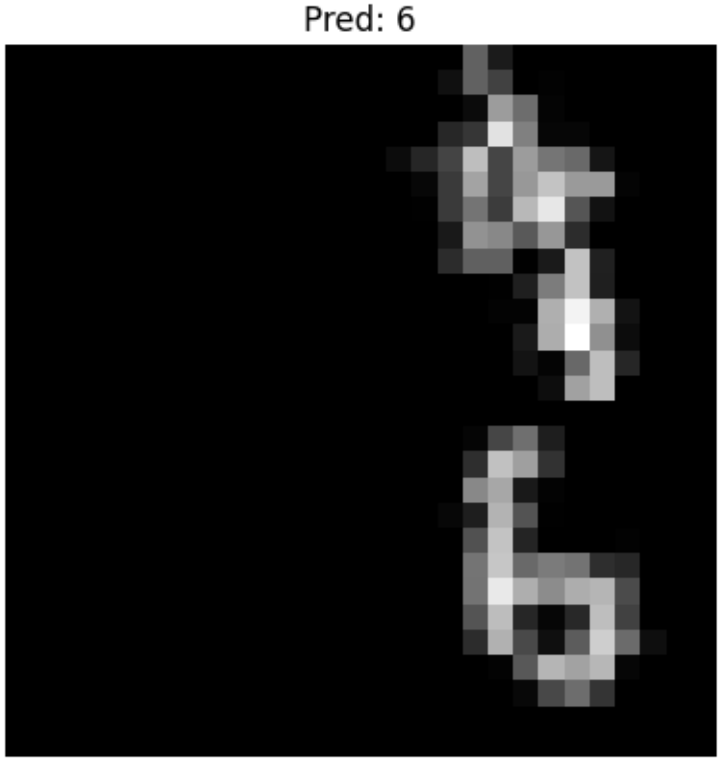

Pred: 8

Pred: 4

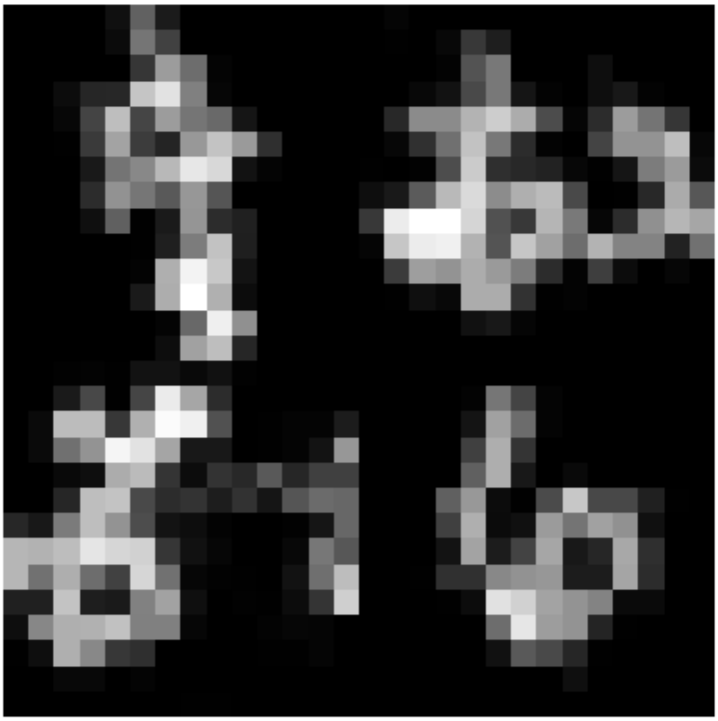


Pred: 9

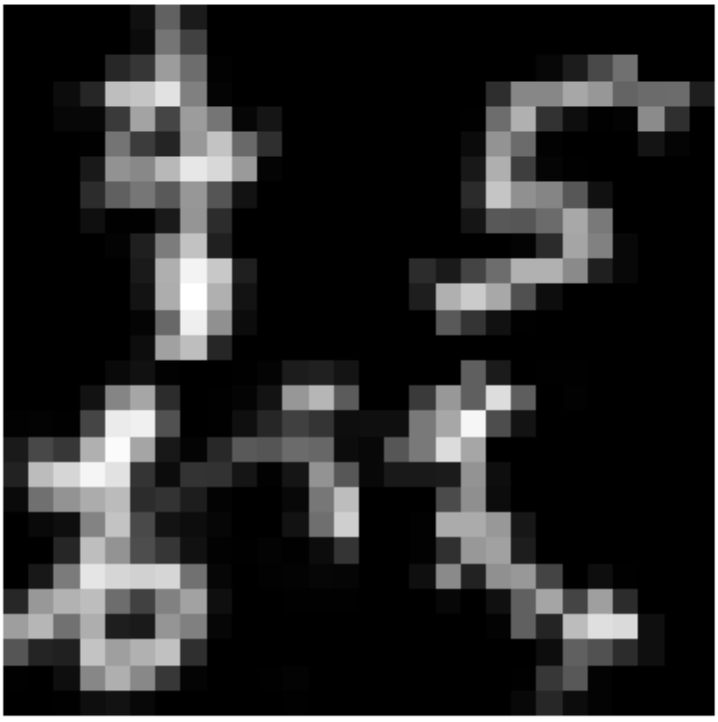


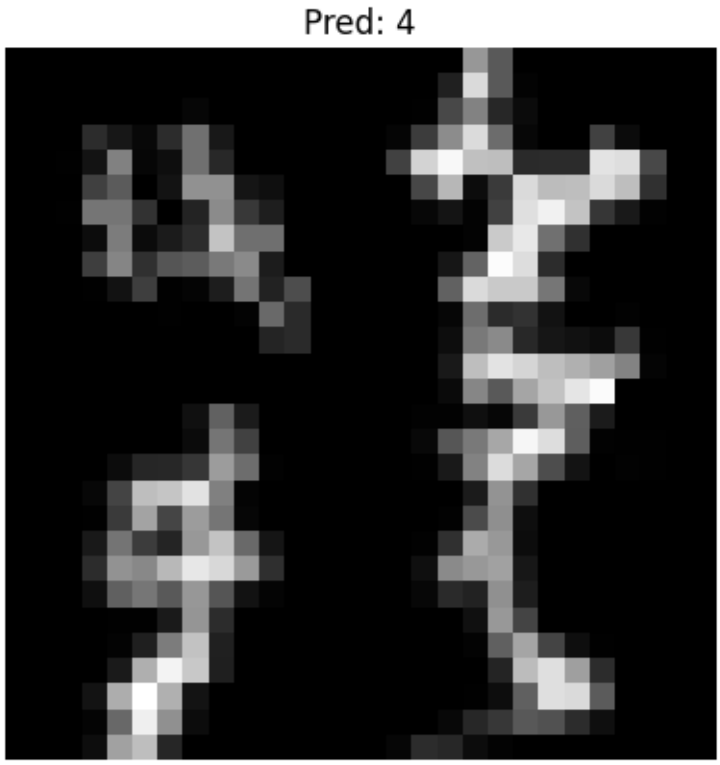

Pred: 6

Pred: 6



Pred: 5





(functions to get code in pdf form)

[10]:
```
!jupyter nbconvert --to pdf  "/content/drive/My Drive/Colab Notebooks/ECSE551/
↪ECSE551Assignment3"
```

```
[NbConvertApp] WARNING | pattern '/content/drive/My Drive/Colab
Notebooks/ECSE551/ECSE551Assignment3' matched no files
This application is used to convert notebook files (*.ipynb)
        to various other formats.

        WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.

Options
=======
The options below are convenience aliases to configurable class-options,
as listed in the "Equivalent to" description-line of the aliases.
To see all configurable class-options for some <cmd>, use:
    <cmd> --help-all

--debug
```

```
    set log level to logging.DEBUG (maximize logging output)
    Equivalent to: [--Application.log_level=10]
--show-config
    Show the application's configuration (human-readable format)
    Equivalent to: [--Application.show_config=True]
--show-config-json
    Show the application's configuration (json format)
    Equivalent to: [--Application.show_config_json=True]
--generate-config
    generate default config file
    Equivalent to: [--JupyterApp.generate_config=True]
-y
    Answer yes to any questions instead of prompting.
    Equivalent to: [--JupyterApp.answer_yes=True]
--execute
    Execute the notebook prior to export.
    Equivalent to: [--ExecutePreprocessor.enabled=True]
--allow-errors
    Continue notebook execution even if one of the cells throws an error and
include the error message in the cell output (the default behaviour is to abort
conversion). This flag is only relevant if '--execute' was specified, too.
    Equivalent to: [--ExecutePreprocessor.allow_errors=True]
--stdin
    read a single notebook file from stdin. Write the resulting notebook with
default basename 'notebook.*'
    Equivalent to: [--NbConvertApp.from_stdin=True]
--stdout
    Write notebook output to stdout instead of files.
    Equivalent to: [--NbConvertApp.writer_class=StdoutWriter]
--inplace
    Run nbconvert in place, overwriting the existing notebook (only
            relevant when converting to notebook format)
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=]
--clear-output
    Clear output of current file and save in place,
            overwriting the existing notebook.
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=
--ClearOutputPreprocessor.enabled=True]
--coalesce-streams
    Coalesce consecutive stdout and stderr outputs into one stream (within each
cell).
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=
--CoalesceStreamsPreprocessor.enabled=True]
--no-prompt
    Exclude input and output prompts from converted document.
```

```
    Equivalent to: [--TemplateExporter.exclude_input_prompt=True
--TemplateExporter.exclude_output_prompt=True]
--no-input
    Exclude input cells and output prompts from converted document.
            This mode is ideal for generating code-free reports.
    Equivalent to: [--TemplateExporter.exclude_output_prompt=True
--TemplateExporter.exclude_input=True
--TemplateExporter.exclude_input_prompt=True]
--allow-chromium-download
    Whether to allow downloading chromium if no suitable version is found on the
system.
    Equivalent to: [--WebPDFExporter.allow_chromium_download=True]
--disable-chromium-sandbox
    Disable chromium security sandbox when converting to PDF..
    Equivalent to: [--WebPDFExporter.disable_sandbox=True]
--show-input
    Shows code input. This flag is only useful for dejavu users.
    Equivalent to: [--TemplateExporter.exclude_input=False]
--embed-images
    Embed the images as base64 dataurls in the output. This flag is only useful
for the HTML/WebPDF/Slides exports.
    Equivalent to: [--HTMLExporter.embed_images=True]
--sanitize-html
    Whether the HTML in Markdown cells and cell outputs should be sanitized..
    Equivalent to: [--HTMLExporter.sanitize_html=True]
--log-level=<Enum>
    Set the log level by value or name.
    Choices: any of [0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR',
'CRITICAL']
    Default: 30
    Equivalent to: [--Application.log_level]
--config=<Unicode>
    Full path of a config file.
    Default: ''
    Equivalent to: [--JupyterApp.config_file]
--to=<Unicode>
    The export format to be used, either one of the built-in formats
            ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook',
'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides', 'webpdf']
            or a dotted object name that represents the import path for an
            ``Exporter`` class
    Default: ''
    Equivalent to: [--NbConvertApp.export_format]
--template=<Unicode>
    Name of the template to use
    Default: ''
    Equivalent to: [--TemplateExporter.template_name]
--template-file=<Unicode>
```

```
    Name of the template file to use
    Default: None
    Equivalent to: [--TemplateExporter.template_file]
--theme=<Unicode>
    Template specific theme(e.g. the name of a JupyterLab CSS theme distributed
    as prebuilt extension for the lab template)
    Default: 'light'
    Equivalent to: [--HTMLExporter.theme]
--sanitize_html=<Bool>
    Whether the HTML in Markdown cells and cell outputs should be sanitized.This
    should be set to True by nbviewer or similar tools.
    Default: False
    Equivalent to: [--HTMLExporter.sanitize_html]
--writer=<DottedObjectName>
    Writer class used to write the
                                        results of the conversion
    Default: 'FilesWriter'
    Equivalent to: [--NbConvertApp.writer_class]
--post=<DottedOrNone>
    PostProcessor class used to write the
                                        results of the conversion
    Default: ''
    Equivalent to: [--NbConvertApp.postprocessor_class]
--output=<Unicode>
    Overwrite base name use for output files.
                Supports pattern replacements '{notebook_name}'.
    Default: '{notebook_name}'
    Equivalent to: [--NbConvertApp.output_base]
--output-dir=<Unicode>
    Directory to write output(s) to. Defaults
                                    to output to the directory of each notebook.
To recover
                                    previous default behaviour (outputting to the
current
                                    working directory) use . as the flag value.
    Default: ''
    Equivalent to: [--FilesWriter.build_directory]
--reveal-prefix=<Unicode>
    The URL prefix for reveal.js (version 3.x).
            This defaults to the reveal CDN, but can be any url pointing to a
copy
            of reveal.js.
            For speaker notes to work, this must be a relative path to a local
            copy of reveal.js: e.g., "reveal.js".
            If a relative path is given, it must be a subdirectory of the
            current directory (from which the server is run).
            See the usage documentation
            (https://nbconvert.readthedocs.io/en/latest/usage.html#reveal-js-
```

```
html-slideshow)
            for more details.
    Default: ''
    Equivalent to: [--SlidesExporter.reveal_url_prefix]
--nbformat=<Enum>
    The nbformat version to write.
            Use this to downgrade notebooks.
    Choices: any of [1, 2, 3, 4]
    Default: 4
    Equivalent to: [--NotebookExporter.nbformat_version]

Examples
--------

    The simplest way to use nbconvert is

            > jupyter nbconvert mynotebook.ipynb --to html

            Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown',
'notebook', 'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides',
'webpdf'].

            > jupyter nbconvert --to latex mynotebook.ipynb

            Both HTML and LaTeX support multiple output templates. LaTeX
includes
            'base', 'article' and 'report'.  HTML includes 'basic', 'lab' and
            'classic'. You can specify the flavor of the format used.

            > jupyter nbconvert --to html --template lab mynotebook.ipynb

            You can also pipe the output to stdout, rather than a file

            > jupyter nbconvert mynotebook.ipynb --stdout

            PDF is generated via latex

            > jupyter nbconvert mynotebook.ipynb --to pdf

            You can get (and serve) a Reveal.js-powered slideshow

            > jupyter nbconvert myslides.ipynb --to slides --post serve

            Multiple notebooks can be given at the command line in a couple of
            different ways:

            > jupyter nbconvert notebook*.ipynb
            > jupyter nbconvert notebook1.ipynb notebook2.ipynb
```

```
or you can specify the notebooks list in a config file, containing::

    c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

> jupyter nbconvert --config mycfg.py

To see all available configurables, use `--help-all`.
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```