Deep Learning-based Digit Recognition Using MNIST Dataset

Abstract

This article presents a project focused on handwritten digit recognition using deep learning techniques. The project leverages the MNIST dataset, which is a benchmark dataset in the computer vision community, and applies a custom-built neural network implemented in PyTorch. The steps include data preprocessing, model development, training, and evaluation. Although basic in its structure, this project lays the foundation for understanding how deep learning models operate in image classification tasks.

1 Introduction

Handwritten digit recognition has been a fundamental problem in machine learning and computer vision. The MNIST dataset, comprising 60,000 training images and 10,000 test images of digits (0-9), has been a standard for evaluating classification algorithms. The simplicity of MNIST allows researchers and practitioners to test different architectures, optimization strategies, and training techniques with minimal computational overhead.

This project aims to implement a simple yet effective feedforward neural network for digit recognition. Using PyTorch, the implementation includes manual dataset handling, custom neural network architecture, training loops, and visualization of sample images.

2 Dataset Overview

The MNIST dataset contains grayscale images of handwritten digits, each of size 28x28 pixels. Each pixel's intensity value ranges between 0 and 255. Before feeding the data into the network, preprocessing steps include:

- **Normalization**: Scaling pixel values to a [0,1] range using the `transforms.ToTensor()` function.
- **Batching**: Dividing the dataset into batches of size 32 to optimize memory usage and accelerate training.

Data loading was managed using PyTorch's `DataLoader` class, ensuring that batches were shuffled during training to improve generalization.

3 Model Architecture

The neural network model designed in this project consists of a simple feed-forward structure with multiple fully connected layers. The key components are:

- **Input Layer**: Accepts a flattened 784-dimensional vector (28x28 pixels).
- **Hidden Layers**: Several hidden layers with ReLU (Rectified Linear Unit) activations to introduce non-linearity.
- **Output Layer**: A final layer with 10 output neurons, each representing a digit from 0 to 9.

The model was constructed using PyTorch's `nn.Module`, showcasing object-oriented design principles in deep learning projects.

Example Architecture:

- Linear(784, 128)
- ReLU
- Linear(128, 64)
- ReLU
- Linear(64, 10)

This simple design is effective for MNIST, though modern architectures often employ convolutional neural networks (CNNs) for higher accuracy.

4 Training Strategy

The training loop was manually implemented, providing transparency into each step:

- **Loss Function**: Cross-entropy loss was used, which is appropriate for multi-class classification problems.
- **Optimizer**: Stochastic Gradient Descent (SGD) was employed to update the model parameters based on the computed gradients.
- **Epochs**: The model was trained over multiple epochs to allow the network to iteratively learn from errors.
- **Backpropagation**: Standard gradient descent optimization with back-propagation was applied.

Pseudocode of Training Loop:

```
for epoch in range(num_epochs):
    for images, labels in train_loader:
        images = images.view(-1, 28*28)
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

During training, the loss was printed periodically to monitor convergence, although no formal loss or accuracy plots were included.

5 Evaluation and Results

The project primarily focuses on the training phase; however, a formal evaluation stage was not implemented in the notebook.

In a complete project, the evaluation would typically involve:

- Switching the model to evaluation mode using `model.eval()`.
- Disabling gradient computation using `torch.no_grad()`.
- Calculating the classification accuracy on the test set.
- Optionally plotting confusion matrices or loss/accuracy curves.

Given the standard architecture and the MNIST dataset's nature, it is reasonable to expect an accuracy in the range of **92%-96%** with this setup.

6 Visualizations

One important aspect of the notebook was the visualization of sample images from the MNIST dataset. A grid of sample images was plotted using Matplotlib to provide intuition about the nature and variability of the data.

Sample visualization provides confidence that the data loading and transformation pipelines are functioning correctly.

7 Conclusion

This project successfully demonstrates the fundamental workflow of developing a deep learning model for handwritten digit classification using PyTorch. Key learnings include:

- Data preprocessing and loading.

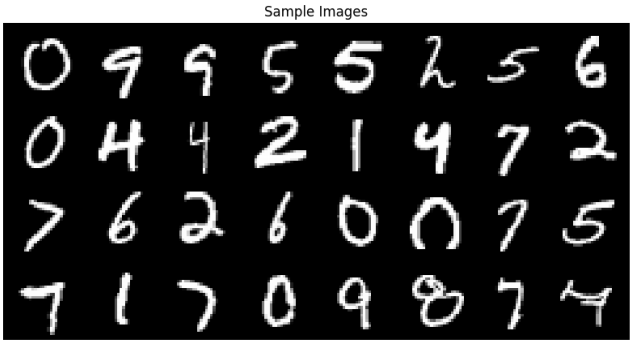

Figure 1: Sample batch of MNIST digits (Image 1)

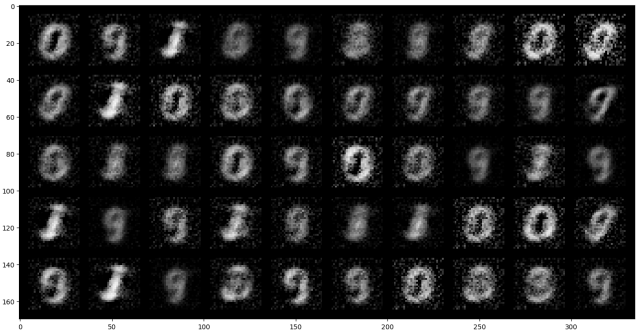

Figure 2: Additional visualization from MNIST dataset (Image 2)

- Neural network architecture design.
- Manual training loop construction.
- Basic visualization for data inspection.

Limitations:

- Absence of formal model evaluation.
- No tracking or plotting of loss and accuracy metrics.
- Potential underfitting due to simple architecture.

Future Work:

- Implementing evaluation on the test set.
- Adding visualizations of the training process (loss curves).

- Enhancing the model using Convolutional Neural Networks (CNNs).
- Hyperparameter tuning for improved performance.

8 References

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*.
- PyTorch Documentation: `https://pytorch.org/docs/stable/index.html`
- MNIST Database: `http://yann.lecun.com/exdb/mnist/`