Fully Connected Neural Networks with NumPy and PyTorch Comparison

Jan Burakowski

1 Introduction

In this project I aim to explore the implementation of a simple fully connected neural network using NumPy, and compare it to an equivalent PyTorch model. The objective is to deepen understanding of core neural network concepts—such as linear transformations, activation functions, and gradient-based optimization—by manually implementing all of the core elements of a Neural Network.

Both models are applied to the MNIST digit classification task, which involves recognizing handwritten digits (0–9) from 28×28 grayscale images. The chosen architecture is intentionally simple: a single hidden layer with 10 neurons, connecting an input of 784 features to 10 output classes.

In this report, I will outline the implementation methodology for both NumPy and PyTorch, analyse experimental results, and compare performance in terms of accuracy and behaviour on different classes.

2 Methodology

2.1 Network Architecture and Data Preparation

Both implementations share a simple architecture: an input layer of 784 features, a hidden layer with 10 ReLU-activated neurons, and an output layer of 10 neurons with softmax activation. The MNIST dataset, containing 60,000 training and 10,000 test images of handwritten digits, is used for classification.

In NumPy, images are flattened and normalized to $[0, 1]$. In PyTorch, I use `torchvision.datasets.MNI` applying a tensor transformation and normalization to $[-1, 1]$. While data formats differ slightly (NumPy arrays vs. PyTorch tensors), both models train on the same inputs.

Training minimizes classification error using gradient descent. The NumPy model applies full-batch updates, while PyTorch uses mini-batch stochastic gradient descent with a batch size of 64. Both use a learning rate of $\alpha = 0.1$, with no regularization applied.

2.2 NumPy Implementation from Scratch

The NumPy model is built from basic components reflecting each mathematical step in a feedforward neural network. Below you will find a summary of the key functions:

`initialize_parameters()` Initializes parameters:

$$W_1 \in \mathbb{R}^{10\times 784}, \quad b_1 \in \mathbb{R}^{10\times 1}, \quad W_2 \in \mathbb{R}^{10\times 10}, \quad b_2 \in \mathbb{R}^{10\times 1}$$

All values are sampled uniformly from $[-0.5, 0.5]$.

`ReLU(x)` Applies $\mathrm{ReLU}(x) = \max(0, x)$ element-wise.

`softmax(x)` Converts logits z_i to probabilities:

$$\mathrm{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

`forward_prop()` Computes activations:

$$Z_1 = W_1 X + b_1, \quad A_1 = \mathrm{ReLU}(Z_1), \quad Z_2 = W_2 A_1 + b_2, \quad A_2 = \mathrm{softmax}(Z_2)$$

`one_hot_encode(y)` Converts labels to one-hot vectors $Y_{\text{one-hot}}$.

`ReLU_deriv(Z)` Computes the derivative of ReLU:

$$\frac{d}{dZ}\mathrm{ReLU}(Z) = \begin{cases} 1 & Z > 0 \\ 0 & Z \leq 0 \end{cases}$$

`backward_prop()` Applies gradients using cross-entropy loss:

$$L = -\sum_k y_k \log A_{2,k}$$

$$\begin{aligned} dZ_2 &= A_2 - Y_{\text{one-hot}} \\ dW_2 &= \frac{1}{m} dZ_2 A_1^T, \quad db_2 = \frac{1}{m}\sum dZ_2 \\ dZ_1 &= W_2^T dZ_2 \circ \mathrm{ReLU\_deriv}(Z_1) \\ dW_1 &= \frac{1}{m} dZ_1 X^T, \quad db_1 = \frac{1}{m}\sum dZ_1 \end{aligned}$$

`update_params()` Updates parameters:

$$W := W - \alpha \cdot \nabla W, \quad b := b - \alpha \cdot \nabla b$$

`get_predictions(A2)` Returns $\arg\max A_2$ per column.

`get_accuracy(pred, Y)` Computes mean accuracy.

`gradient_descent()` Ties everything together:

Listing 1: Training loop (partial) for NumPy implementation.

```
for i in range(iterations):
    Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
    dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2,
       X, Y)
    W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1,
       dW2, db2, learning_rate)

    if i % 100 == 0:
        pred = get_predictions(A2)
        acc = get_accuracy(pred, Y)
        print(f"Iteration␣{i}␣-␣Accuracy:␣{acc:.4f}")
```

This structure mirrors the full forward and backwards pass of a two-layer neural network, allowing parameter updates based on cross-entropy loss and softmax output.

2.3 PyTorch Implementation

To compare my manual NumPy implementation with an industry-standard framework, I recreated the same neural network architecture using PyTorch.

`PyTorchModel` class I defined a class `PyTorchModel` using `torch.nn.Module`, with two layers:

- `fc1 = nn.Linear(784, 10)`: input to hidden layer
- `fc2 = nn.Linear(10, 10)`: hidden to output layer

The forward pass mirrors the NumPy version:

$$x \to \text{ReLU}(W_1 x + b_1) \to \text{softmax}(W_2 h + b_2)$$

Listing 2: Forward pass in the PyTorch model.

```
def forward(self, x):
    x = self.fc1(x)
    x = torch.relu(x)
    x = self.fc2(x)
    return torch.softmax(x, dim=1)
```

`train_pytorch_model()` This function handles the training loop using:

- `nn.CrossEntropyLoss` for loss
- `torch.optim.SGD` for parameter updates

Each batch of images is flattened, passed through the model, and used to compute loss and gradients:

1. `optimizer.zero_grad()`
2. Forward pass: `outputs = model(images)`
3. Loss computation: `loss = criterion(outputs, labels)`
4. Backward pass: `loss.backward()`
5. Parameter update: `optimizer.step()`

Every 100 epochs, I printed the training accuracy to monitor learning progress.

`evaluate_pytorch_model()` After training, I evaluated test accuracy by iterating through batches from `test_loader`, predicting with the trained model, and comparing to true labels. This returned both overall accuracy and class-wise predictions.

`plot_heatmap()` To visualize performance, I used this function to create confusion matrix heatmaps and classification reports. These helped highlight per-class accuracy and any systematic misclassifications in either model.

3 Results and Discussion

I trained both the NumPy and PyTorch implementations on the MNIST dataset using 500 iterations/epochs with a learning rate of 0.1. This section summarizes the observed training behavior, final test accuracies, and class-wise performance.

Training Progress

The learning dynamics varied due to the optimization strategies used:

- **NumPy (Batch Gradient Descent):** Accuracy started at 15% and steadily rose to over 83% by iteration 500. Updates occurred after evaluating the entire dataset each time.
- **PyTorch (Mini-Batch SGD):** Faster early convergence, reaching over 62% accuracy in just one epoch. By epoch 100 it was at 87%, plateauing near 88%. Frequent mini-batch updates improved early learning speed.

While the PyTorch model made significantly more updates due to mini-batching, both models reached a comparable performance range by the end of training.

Test Accuracy and Comparison

Test accuracy on the unseen 10,000-image set was:

- **NumPy:** 83.41%
- **PyTorch:** 85.27%

The PyTorch model's slight edge likely results from more updates and better internal optimization. Its use of mini-batch updates can introduce gradient noise, which sometimes helps generalization. However, both models generalized well given their simple architectures (only 10 hidden units).

Confusion Matrix Analysis

To assess per-class performance, I analyzed confusion matrices and classification reports.

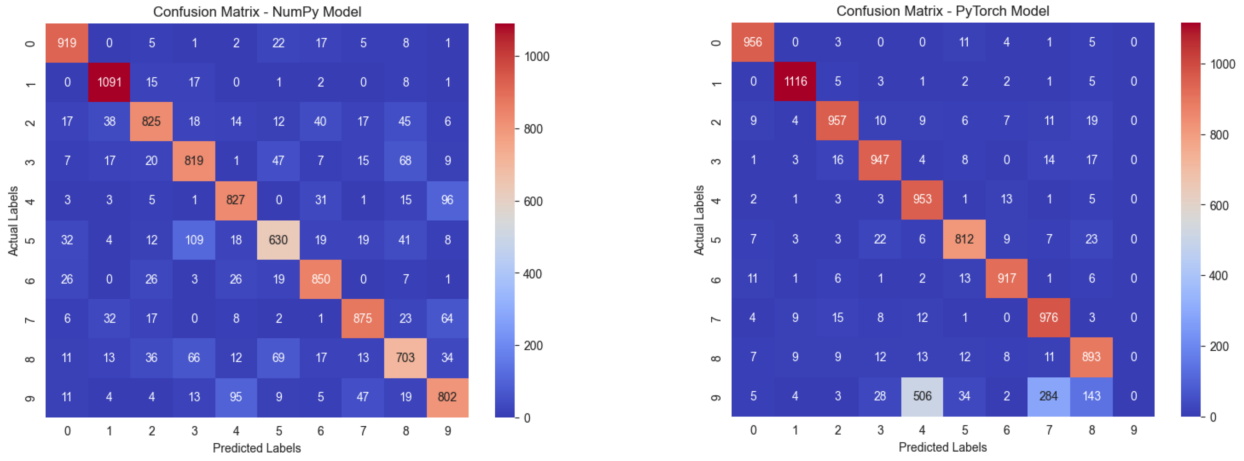

Figure 1: Confusion matrices for NumPy (left) and PyTorch (right) models.

NumPy Model:

- Balanced performance across classes, with precision and recall typically in the 70–96% range.
- Struggled most with digits **5** and **8**, often confusing them with similar-looking digits.
- Achieved a recall of 79% on digit **9**, maintaining reasonable accuracy across the board.

PyTorch Model:

- Achieved excellent performance (above 95% recall) on digits **0**, **1**, **2**, **3**, and **6**.
- Completely failed to predict digit **9**, with 0% precision and recall, likely due to local minima or class neglect during training.
- Despite the failure on class 9, the overall accuracy remained high.

These results highlight a key insight: higher accuracy does not always equate to better class-wise performance. The NumPy model, though slightly less accurate, demonstrated more consistent behavior across all classes.

Conclusion of Results

This comparison demonstrated that both the manual NumPy approach and the PyTorch implementation effectively learn to classify handwritten digits. The NumPy model provided full transparency into the learning process, while the PyTorch version reflected a modern machine learning approach.

Key takeaways:

- PyTorch achieved better accuracy through more frequent updates.
- The NumPy model offered balanced predictions without catastrophic failure.
- Misclassifications (especially PyTorch's treatment of class 9) shows the importance of monitoring per-class performance, not just overall accuracy.

This experiment reinforced my understanding of the mathematical foundations of neural networks and their practical application using industry tools like PyTorch. It also demonstrated how implementation choices impact performance characteristics and debugging visibility.