MNIST Classifier
Artificial Neural Networks
Project Documentation

Anton Dmitriev
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
`xdmitriev@stuba.sk`

Contents

1. Introduction
This project focuses on developing a neural network-based classifier to recognize handwritten digits from the MNIST dataset. The MNIST dataset contains big database of grayscale images, each representing digits from 0 to 9. The classifier is trained using three optimization algorithms with the goal of achieving a test accuracy greater than 97%.

2. Task
Create a feedforward neural network (multilayer perceptron) to classify handwritten digits from the MNIST dataset. Train the model using three optimizers: SGD, SGD + Momentum, and ADAM. Measure training and test loss, as well as accuracy (percentage of correct predictions on the test set). Aim for accuracy above 97%. Use `PyTorch` for implementation.

2.1. Steps
1. **Dataset:** Load MNIST using `PyTorch`, split into training and test sets, and normalize values to `[0, 1]`.
2. **Model Design:** Define the network architecture with chosen layer sizes and activation functions.
3. **Hyperparameters:** Configure learning rate, batch size, and number of epochs.
4. **Training:** Train the model for the same number of epochs with each optimizer.
5. **Evaluation:** Compare losses, accuracy, and confusion matrices for all models.

3. Structure

3.1. classifier.py
The main script that initializes the neural network, loads the dataset, and manages training and evaluation. It imports configurations from `'config.py'` and uses utility functions to train the model and plot metrics. The script performs training with three optimizers (SGD, SGD + Momentum, and ADAM) and saves results for comparison.

Key Functions:
- `train_and_evaluate()`: Trains the model and computes train/test losses and accuracy.
- `plot_metrics()`: Visualizes loss and accuracy trends during training and testing.
- `plot_confusion_matrix()`: Generates a confusion matrix to evaluate model performance.

3.2. config.py
Contains configuration parameters and hyperparameters, including:
- **Dataset Configuration:** `train_batch_size`, `test_batch_size`
- **Neural Network Configuration:** `second_layer_in`, `second_layer_out`
- **Training Configuration:** `epochs`, `sgd_learning_rate`, `sgd_momentum_learning_rate`, `adam_learning_rate`, `momentum`

4. How to Run

4.1. Run Main Script
To train and evaluate the model, run the `classifier.py` script:

```
python classifier.py
```

4.2. Adjust Parameters
Modify `'config.py'` to update model settings and hyperparameters.

4.3. Dependencies
The project requires the following Python libraries:

- **torch** for model implementation and training
- **torchvision** for dataset loading and transformations
- **matplotlib** and **seaborn** for visualization
- **scikit-learn** for generating confusion matrix

Install dependencies with:

```
pip install torch torchvision matplotlib seaborn scikit-learn
```

5. Architecture

5.1. Dataset

The MNIST dataset is used for training and testing. It consists of 60,000 training images and 10,000 testing images, each representing a digit from 0 to 9. Images are grayscale with a resolution of 28×28:

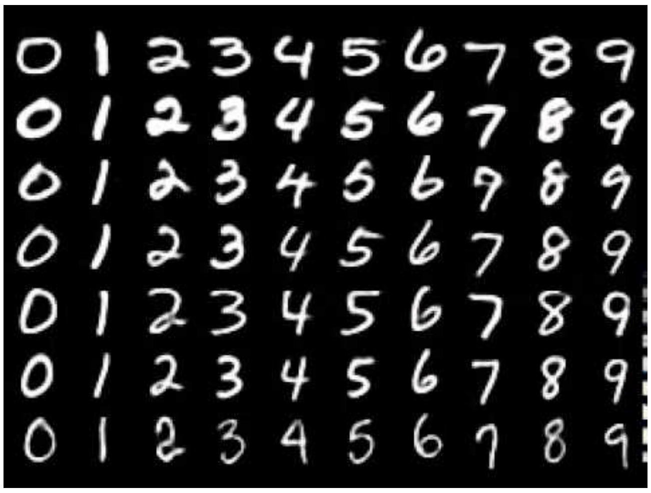

Figure 1: MNIST Dataset Exsample

5.1.1. Normalization

- **Tensor Convertion:** Converts each image into a tensor with pixel values scaled to the range `[0,1]` because models operate with tensors.
- **Normalization:** Applies normalization to adjust pixel values to the range `[-1,1]`, using the formula `Normalized Value = (Pixel Value - Mean) / Standard Deviation`, ensuring the dataset is also centered around 0.5.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

5.1.2. Types

```
train_dataset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
])
```

- **Data Loading:** Dataset is directly downloaded via `torchvision.datasets`, ensuring accessibility without including it in the project.

- **Training Set:** Dataset of 60,000 images in total because is used to train the model.
- **Testing Set:** Dataset of 10,000 images in total because is used only to evaluate the model's - performance on unseen data.
- **Transformation:** Applies the defined before transformations to preprocess the images.

5.1.3. Batches

- **Batch Size:** Splits data into groups(batches) by selected size, that determines how many images are processed at once.
- **Shuffling:**
  - **Training:** Shuffling ensures that at each epoch the data is in a different order so that the model learns to recognize the numbers, rather than being tricky by memorizing their ordering.
  - **Testing:** Shuffling is disabled because it is unnecessary to ensure consistent evaluation.

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

5.2. Neural Net

The model is a simple feedforward neural net designed especially for classifying MNIST digits. It consists of three fully connected layers:

- **Input Layer:** Accepts flattened input images of size 28 × 28 = 784.
- **Support Layer** Contains as input `second_layer_in` (128) and as output `second_layer_out` neurons (64).
- **Output Layer:** Contains 10 neurons, one for each digit class (0-9).

```
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(28 * 28, second_layer_in)
        self.fc2 = nn.Linear(second_layer_in, second_layer_out)
        self.fc3 = nn.Linear(second_layer_out, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- **Activation Function:** ReLU (Rectified Linear Unit) become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance

5.3. Loss Function

- **Loss Function:** This function measures how far the model's predictions are from the actual target values.
- **Cross-Entropy:** In this project, the loss function used is Cross-Entropy Loss Function, because it is specifically designed to handle multi-class classification tasks like MNIST digit recognition.

```
criterion = nn.CrossEntropyLoss()
```

6. Training

The training process is handled by the `train_and_evaluate()` function, which performs both training and evaluation in each epoch:

```python
def train_and_evaluate(optimizer, epochs=5):
    train_losses = []
    test_losses = []
    test_accuracies = []

    for epoch in range(epochs):
        # training
        model.train()
        running_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        train_losses.append(running_loss / len(train_loader))

        # evaluating
        model.eval()
        test_loss = 0.0
        correct = 0
        total = 0
        with torch.no_grad():
            for images, labels in test_loader:
                outputs = model(images)
                loss = criterion(outputs, labels)
                test_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        test_losses.append(test_loss / len(test_loader))
        accuracy = 100 * correct / total
        test_accuracies.append(accuracy)

        print(f"Epoch {epoch + 1}/{epochs}, Train Loss {train_losses[-1]:.4f}, Test
Loss {test_losses[-1]:.4f}, Accuracy {accuracy:.2f}%")

    return train_losses, test_losses, test_accuracies
```

1. **Training Phase:**
   - **Training Mode:** `model.train()` ensures layers like dropout or batch normalization behave correctly during training.
   - **Clear Gradients:** `optimizer.zero_grad()` prevents accumulation of gradients from previous steps.
   - **Forward Pass:** Compute outputs by passing images through the model.
   - **Loss Calculation:** Compute the difference between predictions and true labels using `criterion`.
   - **Backward Pass:** Propagate errors back through the network to compute gradients.
   - **Weight Update:** `optimizer.step()` adjusts model parameters using the computed gradients.
2. **Evaluation Phase:**

- **Evaluation Mode:** `model.eval()` ensures consistent behavior, such as disabling dropout.
- **No Gradient Computation:** `torch.no_grad()` reduces computation overhead during evaluation.
- **Accuracy Calculation:** Compare predictions with true labels and compute the percentage of correct predictions.

3. **Metrics Tracking:**
   - **Train Loss:** Average training loss across batches.
   - **Test Loss:** Average test loss across batches.
   - **Accuracy:** Percentage of correctly classified samples in the test set.

7. Visualization

7.1. Training Metrics

This function generates two graphs:
- A loss graph for the training and test sets
- The accuracy graph on the test set

```
def plot_metrics(train_losses, test_losses, test_accuracies, title):

    # plot losses
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label="Train Loss")
    plt.plot(epochs, test_losses, label="Test Loss")
    ...

    # plot accuracy
    plt.subplot(1, 2, 2)
    plt.plot(epochs, test_accuracies, label="Accuracy")
    ...

    plt.show()
```

- **Purpose:** Shows model improvement over epochs.

7.2. Confusion Matrix

This function builds a heatmap of the error matrix showing how often the model correctly or incorrectly classifies each digit.

```
def plot_confusion_matrix(title):
    true_values = []
    predicted_values = []
    model.eval()
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            true_values.extend(labels.tolist())
            predicted_values.extend(predicted.tolist())

    cm = confusion_matrix(true_values, predicted_values)
    ...

    plt.show()
```

- **Purpose:** Evaluates model performance on individual classes.

8. Comparison

Below are the results of training the model with three optimizers: SGD, SGD with Momentum, and Adam, over 25 epochs. This number of epochs was chosen because it is the lowest number of epochs at which each model achieves the goal of overcoming 97.00% accuracy rate. Other configuration parameters at which models reach such a result can be found in `'config.py'` file. Key observations are drawn for each optimizer:

8.1. Train Comparison

8.1.1. Training with SGD

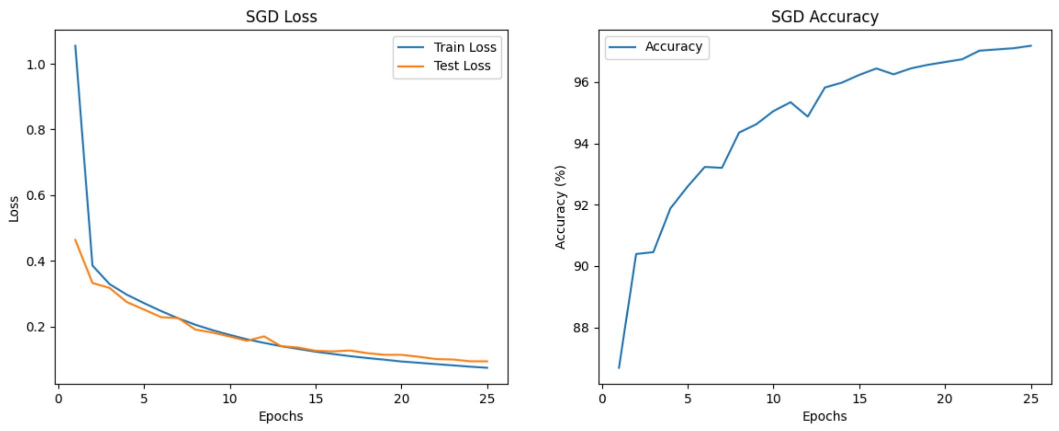

Figure 2: SGD Metrics

```
...
Epoch 20/25, Train Loss 0.0938, Test Loss 0.1139, Accuracy 96.65%
Epoch 21/25, Train Loss 0.0899, Test Loss 0.1082, Accuracy 96.74%
Epoch 22/25, Train Loss 0.0859, Test Loss 0.1012, Accuracy 97.02%
Epoch 23/25, Train Loss 0.0820, Test Loss 0.0997, Accuracy 97.06%
Epoch 24/25, Train Loss 0.0779, Test Loss 0.0944, Accuracy 97.10%
Epoch 25/25, Train Loss 0.0746, Test Loss 0.0941, Accuracy 97.18%
```

Observation:

- The training loss consistently decreases, indicating effective learning.
- The test accuracy improves steadily and achieves 97.18%, meeting the goal of the task.
- However, the convergence is slower compared to other optimizers, and achieves accuracy greater than 97.00% only in epoch 22.

8.1.2. Training with SGD + Momentum

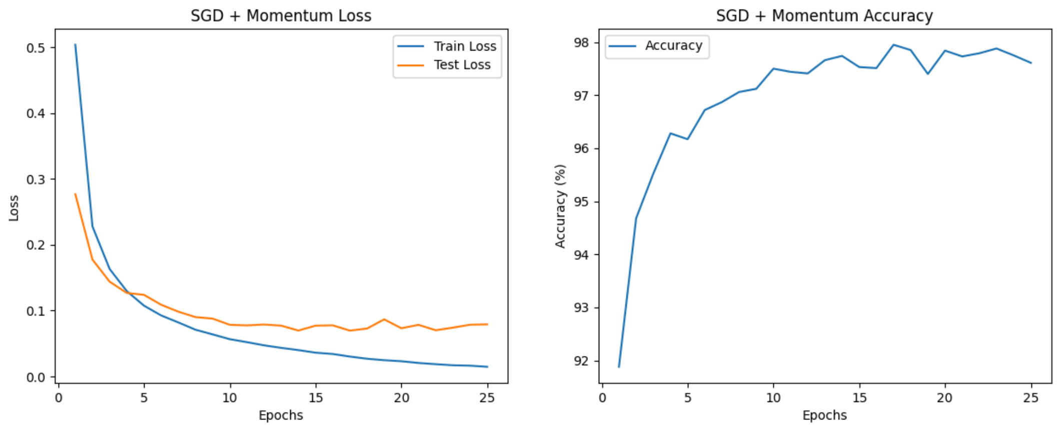

Figure 3: SGD + Momentum Metrics

```
...
Epoch 6/25, Train Loss 0.0927, Test Loss 0.1090, Accuracy 96.72%
Epoch 7/25, Train Loss 0.0821, Test Loss 0.0984, Accuracy 96.87%
Epoch 8/25, Train Loss 0.0710, Test Loss 0.0902, Accuracy 97.06%
Epoch 9/25, Train Loss 0.0639, Test Loss 0.0879, Accuracy 97.12%
...
Epoch 16/25, Train Loss 0.0342, Test Loss 0.0777, Accuracy 97.51%
Epoch 17/25, Train Loss 0.0302, Test Loss 0.0697, Accuracy 97.95%
Epoch 18/25, Train Loss 0.0269, Test Loss 0.0728, Accuracy 97.85%
...
Epoch 24/25, Train Loss 0.0164, Test Loss 0.0786, Accuracy 97.75%
Epoch 25/25, Train Loss 0.0147, Test Loss 0.0792, Accuracy 97.61%
```

Observation:
- Adding momentum increases the convergence speed of the SGD optimizer.
- The test accuracy achieves 97.95%, outperforming SGD.
- The accuracy requirement of 97.00% is achieved as early as epoch 8.
- The model reaches lower losses in fewer epochs, showing that momentum accelerates updates in relevant directions.
- However, although momentum improves learning dynamics, the highest accuracy of 97.95% does not persist, as there is a decrease in accuracy in later epochs. These fluctuations highlight the variability introduced by momentum.

8.1.3. Training with ADAM

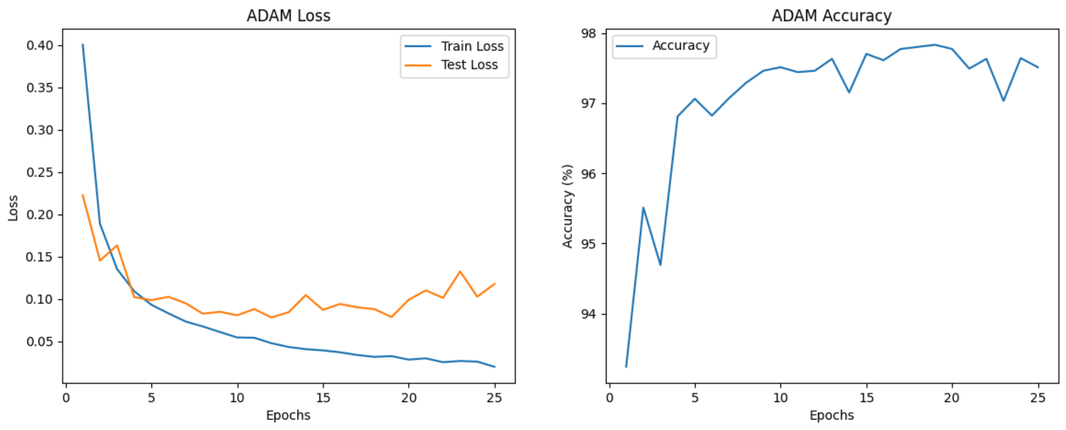

Figure 4: ADAM Metrics

```
...
Epoch 3/25, Train Loss 0.1352, Test Loss 0.1631, Accuracy 94.69%
Epoch 4/25, Train Loss 0.1089, Test Loss 0.1020, Accuracy 96.81%
Epoch 5/25, Train Loss 0.0929, Test Loss 0.0983, Accuracy 97.06%
Epoch 6/25, Train Loss 0.0827, Test Loss 0.1024, Accuracy 96.82%
...
Epoch 17/25, Train Loss 0.0335, Test Loss 0.0899, Accuracy 97.77%
Epoch 18/25, Train Loss 0.0313, Test Loss 0.0878, Accuracy 97.80%
Epoch 19/25, Train Loss 0.0322, Test Loss 0.0784, Accuracy 97.83%
...
Epoch 24/25, Train Loss 0.0257, Test Loss 0.1025, Accuracy 97.64%
Epoch 25/25, Train Loss 0.0195, Test Loss 0.1175, Accuracy 97.51%
```

Observation:
- ADAM achieves the target accuracy of 97.00% as early as epoch 5, outperforming both SGD and SGD + Momentum in early-stage convergence.
- The peak accuracy of 97.83% is reached at epoch 19. However, this is slightly below the highest accuracy achieved by SGD + Momentum (97.95%).
- ADAM shows less fluctuation in accuracy compared to SGD + Momentum, but does not retain peak accuracy in later epochs and its final result 97.51% is slightly worse than SGD + Momentum (97.61%)
- While ADAM is effective in achieving fast convergence and stable training, its overall performance does not surpass SGD + Momentum, which achieves higher peak accuracy.

8.2. Graphical Comparison

Loss curves:
- Training and testing losses decrease consistently for all optimizers, but SGD + Momentum achieves the lowest losses.
- The graphs for SGD are the smoothest, showing its stability in contrast to the other models.
- Also the SGD training/test graph has the least deviation from each other.
- ADAM converges faster at early epochs, but shows more fluctuations at later epochs than all other models.

Curves of Accuracy:.
- ADAM and SGD + Momentum achieve higher accuracy than SGD.
- ADAM reaches peak accuracy earlier, but SGD + Momentum retains slightly better accuracy in later epochs.
- SGD here also shows the least deviation and stability.

8.3. Confusion Matrix Comparison

- The confusion matrix for each optimizer shows how accurately the model predicts each digit class (0-9).
- The diagonal values represent correct predictions.

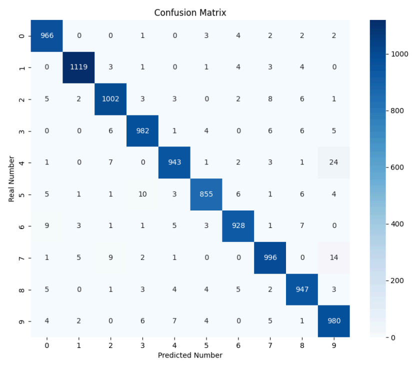

Figure 5: SGD Confusion Matrix

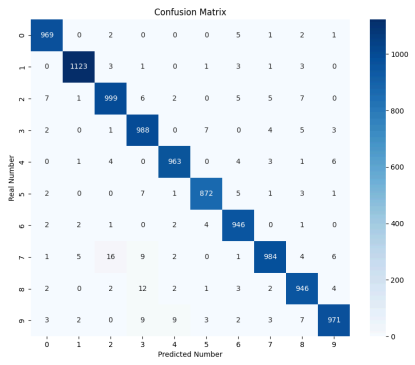

Figure 6: SGD + Momentum Confusion Matrix

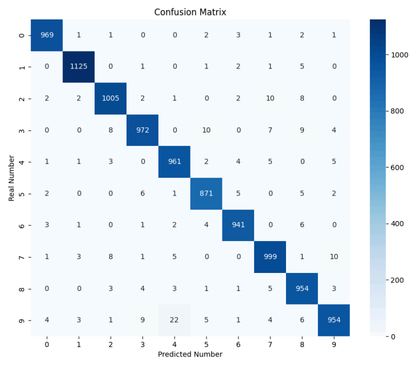

Figure 7: ADAM Confusion Matrix

Insights:

- The best guessed number is 1 while the worst guessed number is 5
- Most errors occur between digits with similar shapes (e.g., 2 and 7, 4 and 9).
- SGD + Momentum has the fewest misclassifications, reflecting its better generalization.

8.4. Summary

Optimizer	First Reached Epoch	Best Accuracy	Last Accuracy	Last Train Loss	Last Test Loss	Convergence Speed
SGD	22	97.18%	97.18%	0.0746	0.0941	Slow
SGD + Momentum	8	97.95%	97.61%	0.0147	0.0792	Faster
ADAM	5	97.80%	97.51%	0.0195	0.1175	Fastest

Table 1: Optimizer Comparison

9. Conclusion

Experiments show that all three optimization algorithms are able to achieve the desired accuracy for the MNIST classification task. However, their performance differs in terms of convergence speed, stability and final accuracy:

- **SGD:** robust but slow convergence, suitable for tasks where stability is prioritized over speed.
- **SGD + Momentum:** The most efficient optimizer, combining fast convergence with high peak accuracy and good generality.
- **ADAM:** Ideal for fast convergence, especially in the early stages of training, but somewhat less effective in terms of final accuracy.

Overall, **SGD + Momentum** is the best choice for this task, striking a balance between training speed and model performance.