Report for Assignment 3 - MLP

Submitted by Rahaf Sbeh & Saar Buium

Objective

The purpose of this assignment is to extend the code provided in Chapter 11 of the book "Machine Learning with PyTorch and Scikit-Learn" by Raschka et al. (2022) to address two hidden layers instead of only one layer.
The assignment required:

1. Modifying the original implementation to include two hidden layers.
2. Classifying the handwritten digits from the MNIST dataset using this modified architecture.
3. Evaluating the prediction performance using macro AUC under a 70%-30% train-test split.
4. Comparing the results with the original single-hidden-layer implementation and a fully connected ANN implemented in PyTorch.

Solution

Extending the implementation for 2-hidden layers

The original implementation was modified to include two hidden layers in the fully connected ANN architecture. The changes included:

1. **Modifications to the Constructor**
   In the constructor an additional set of weights and biases was added to define the second hidden layer, set of weights (weight_h_2) and biases (bias_h_2) was introduced to connect the first hidden layer to the second hidden layer.
   The initialization of these parameters followed the same methodology as the original, random weights with a normal distribution (loc=0.0, scale=0.1) and biases initialized to zeros to ensure consistent scaling of the parameters.

2. **Forward pass**
   Adding an additional hidden layer in the forward propagation function. The forward pass now includes computations for the second hidden layer in addition to the input-to-hidden and hidden-to-output computations.

   Input to First Hidden Layer
   - Linear transformation to first hidden layer computed as before $z_h = X \cdot W_h^T + b_h$
   - Sigmoid activation function as before $a_h = \sigma(z_h)$

First hidden layer to second hidden layer

- The second hidden layer receives as input the output of the first hidden layer (a_h)
  $z_h^{(2)} = a_h \cdot W_h^{(2)T} + b_h^{(2)}$
- Sigmoid activation function $a_h^{(2)} = \sigma\left(z_h^{(2)}\right)$

Second hidden layer to the Output layer

- The output layer receives as input the output of the second hidden layer (a_h_2)
  $z_{out} = a_h^{(2)} \cdot W_{out}^T + b_{out}$
- Sigmoid activation function $a_{out} = \sigma(z_{out})$

3. **Backward pass**

Adjusting the backpropagation logic to account for the additional layer.
It computes gradients for the additional parameters introduced by the second hidden layer to ensure proper weight updates during training.

Output layer gradients

The gradients for the output layer weights (d_loss__dw_out) and biases (d_loss__db_out) are computed based on the difference between the network's predictions and the actual target values (delta_out). This step remains like the original implementation but uses the output of the second hidden layer instead of the first.

Second hidden layer

The second hidden layer receives gradients propagated back from the output layer. These gradients are adjusted by considering the activation function applied in the second hidden layer. Using these adjusted gradients, the model updates the weights and biases connecting the first and second hidden layers.

- The second hidden layer processes the gradients passed back from the output layer. These gradients are computed using d_z_out__a_h_2 (the derivative of the output layer's pre-activation values with respect to the activations of the second hidden layer).
- The delta term for the second hidden layer, delta_h_2, represents the gradient of the loss with respect to the activations of the second hidden layer (a_h_2). It accounts for the activation function's derivative, ensuring non-linear transformations are properly adjusted.
- Using delta_h_2, gradients for the weights (d_loss__d_w_h_2) and biases (d_loss__d_b_h_2) of the second hidden layer are computed. These variables correspond to the weights (self.weight_h_2) and biases (self.bias_h_2) connecting the first and second hidden layers.

First hidden layer

The gradients from the second hidden layer are further propagated back to the first hidden layer. These gradients are adjusted to account for the activation function applied in the first hidden layer. This step ensures that the weights and biases between the input layer and the first hidden layer are updated correctly.

- Gradients are propagated further back from the second hidden layer to the first hidden layer using d_z_h_2__a_h (the derivative of the second hidden layer's pre-activation values with respect to the activations of the first hidden layer, represented by a_h).
- The delta term for the first hidden layer, delta_h, reflects the gradient of the loss with respect to the activations of the first hidden layer. It accounts for the activation function's derivative applied at this layer.
- Gradients for the first hidden layer weights (d_loss__d_w_h) and biases (d_loss__d_b_h) are then computed, updating the weights (self.weight_h) and biases (self.bias_h) that connect the input features (x) to the first hidden layer.

The original architecture of the ANN

- Input Layer: 784 neurons (for MNIST's 28x28 pixel images)
- Hidden Layer 1: 50 neurons
- Output Layer: 10 neurons (one for each digit class)

The revised architecture of the ANN

- Input Layer: 784 neurons (for MNIST's 28x28 pixel images)
- Hidden Layer 1: 500 neurons
- Hidden Layer 2: 500 neurons
- Output Layer: 10 neurons (one for each digit class)

Dataset and Preprocessing

All steps we applied were the same as the original code found in ch11.ipynb
The MNIST dataset was used for training and evaluation. The preprocessing steps included:

- Normalizing the pixel values to the range [0, 1].
- Flattening each 28x28 image into a 784-dimensional vector.
- Splitting the dataset into 70% training and 30% testing subsets.

Evaluation of the Two-hidden-layer Model

The two-hidden-layer neural network exhibited exceptional performance, reflecting significant improvements in accuracy, precision, recall, and overall model reliability throughout the training process.

Initial Performance

At the beginning of the training process, the model demonstrated limited predictive capability:

- **Initial Test MSE**: 0.2
- **Initial Test Accuracy**: 9.9%

These results highlight the necessity of training to refine the model's weight parameters and optimize its predictive performance.

Training Progress

The model was trained for 50 epochs, during which substantial improvements were achieved. The training process was characterized by steady declines in the mean squared error (MSE) and corresponding increases in both training and test accuracies.

- **Final Training MSE**: 0.01
- **Final Training Accuracy**: 97.38%
- **Final Test Accuracy**: 96.36%

Key milestones during training included:

1. **Epoch 10**: Test accuracy surpassed 92%, indicating rapid learning during the early stages of training.
2. **Epoch 30**: Test accuracy plateaued near 95%, reflecting diminishing returns from further epochs.
3. **Epoch 50**: Final convergence to a test accuracy of 96.36%.

Test Accuracy and Macro AUC

The model achieved a **test accuracy of 96.36%**, with a remarkable **macro AUC score of 99.67%**, showcasing its capability to distinguish between different classes with near-perfect sensitivity and specificity. These metrics indicate the model's robustness and its ability to generalize well to unseen data.

Detailed Classification Report

The model's performance was further validated using a classification report, which provided precision, recall, and F1-scores for each class:

- **Precision**: Average of 96%, indicating that the model produces highly relevant predictions.

- **Recall**: Average of 96%, demonstrating the model's ability to identify nearly all relevant instances.
- **F1-Score**: Average of 96%, confirming a balance between precision and recall.

Each class was represented with high scores, with slight variations across classes. The precision and recall values consistently exceeded 95%, highlighting the model's reliability across diverse data distributions.

Confusion Matrix Analysis

The confusion matrix provided further insights into the model's classification ability

Num	precision	recall	f1	support
0	0.97	0.98	0.98	2071
1	0.98	0.98	0.98	2363
2	0.97	0.95	0.96	2097
3	0.97	0.94	0.96	2142
4	0.97	0.96	0.96	2047
5	0.95	0.95	0.95	1894
6	0.97	0.97	0.97	2063
7	0.96	0.97	0.96	2188
8	0.95	0.96	0.96	2048
9	0.95	0.96	0.95	2087

- Most classes exhibited minimal misclassification, with true positives significantly outweighing false positives and negatives.
- For example:
    - Class `0` achieved 98% recall, with only 35 misclassified samples out of 2,071.
    - Class `5` faced slightly higher misclassification rates, likely due to overlapping features with neighboring classes.

The model effectively minimized errors across all classes, contributing to its high test accuracy.

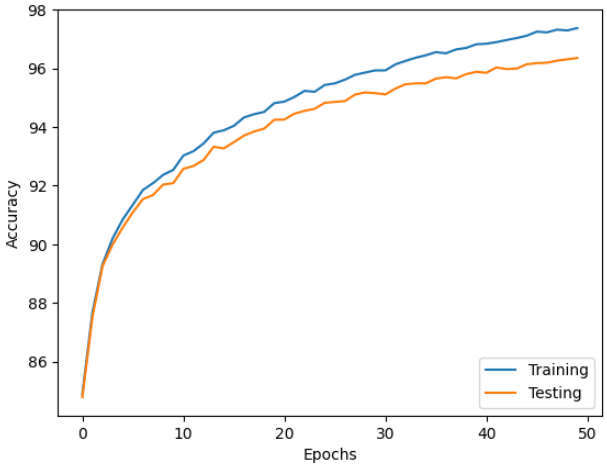
Model Inference Time

The model achieved an inference time of **2.35 seconds** for a dataset containing 21,000 samples. This result demonstrates the model's efficiency, making it suitable for applications where real-time or near-real-time predictions are critical.

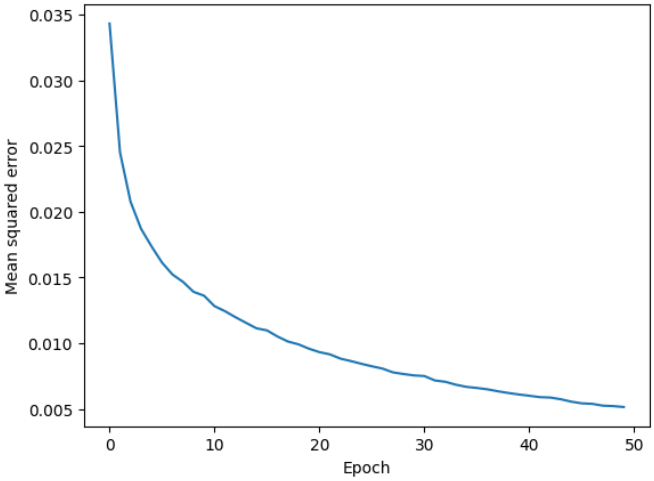
Training Curve and Generalization

Examining the training and test accuracy curves reveals the model's ability to generalize:

- The gap between training and test accuracy remained narrow throughout training, indicating that the model did not overfit to the training data.



- The steady decline in MSE suggests the optimizer effectively minimized the loss function, reinforcing the model's capacity to extract meaningful patterns.



The two-hidden-layer model demonstrated remarkable performance, achieving high accuracy, precision, and recall, while maintaining computational efficiency. Its classification capabilities, supported by strong macro AUC and F1-scores, make it a robust solution for the problem at hand.

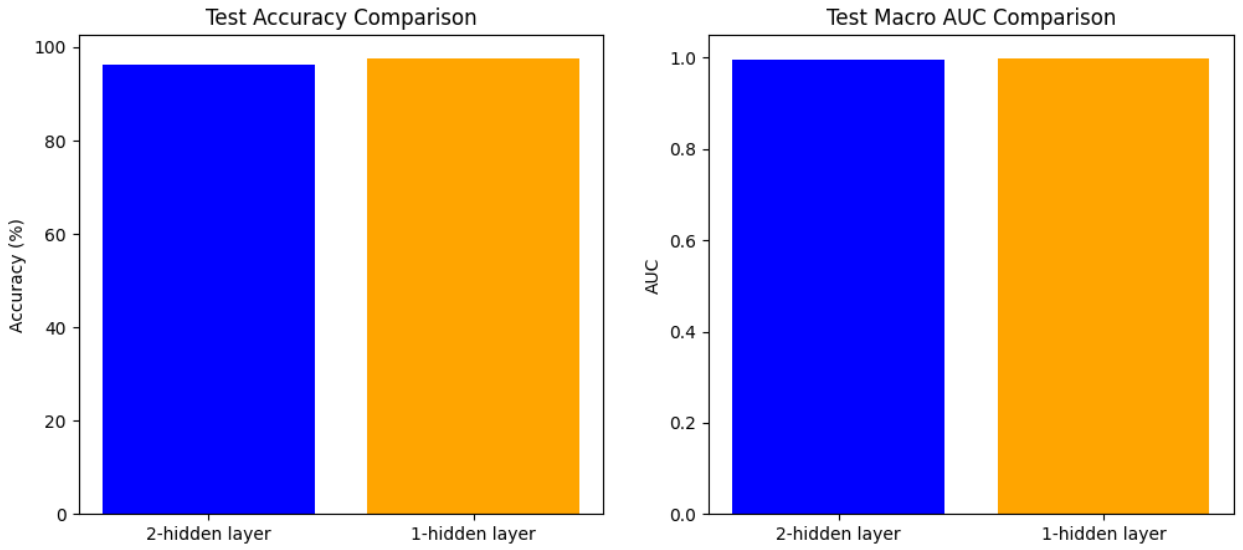
Comparison

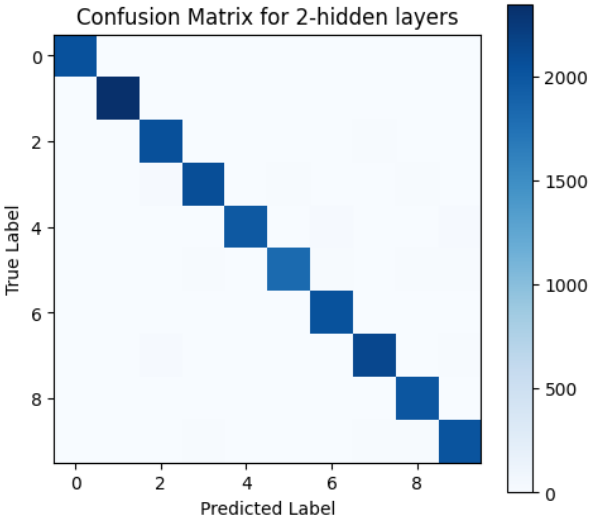
The performance of the two-hidden-layer model was compared with:

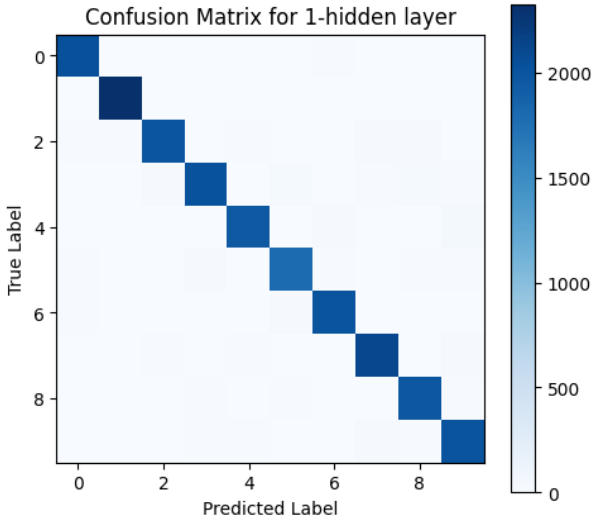
1. The original single-hidden-layer implementation found in ch11.ipynb.(Model 1)
2. A fully connected ANN implemented in PyTorch. (Model 2)

Model Performance

Model	**Accuracy**	**Marco AUC**
Model 1	0.9454	NA
2-hidden layers	0.9636	0.9967
Model 2	**0.9768**	**0.9995**







Observation and Conclusion

Model 2 (with 1 hidden layer using pyTorch) achieves higher test accuracy (97.68% vs. 96.36%) and a better Macro AUC (0.9995 vs. 0.9967) compared to 2-hidden layers model. This indicates that a simpler architecture (1-hidden layer) is more effective in this scenario, offering both higher accuracy and improved class discrimination.

The results suggest that adding more hidden layers does not necessarily improve performance and might introduce unnecessary complexity. One hidden layer using pyTorch is the preferred choice for its slightly better performance and efficiency from other 2 models.

The extension of the original single-hidden-layer ANN to two hidden layers resulted in improved classification performance on the MNIST dataset. The macro AUC metric demonstrated that deeper architectures can enhance the model's ability to capture complex patterns in the data.

However, the fully connected ANN implemented in PyTorch achieved the best performance, suggesting that modern libraries provide additional benefits in terms of optimization and scalability.

GitHub Link
https://github.com/rahafsb/Assignment3_209092196_208994616