Assignment No. 3: Extending and Evaluating a Multi-layer Artificial Neural Network

- **Name:** Adir Serruya
- **Student ID:** 316472455

- **Submission Date:** 2/1/2025

1. Objective

- Extending the existing neural network implementation to include two hidden layers.
- Applying the extended model to classify handwritten digits from the MNIST dataset.
- Evaluating and comparing the performance with the original single hidden layer model and a PyTorch-based ANN implementation.

2. Methodology

1. Extending the Neural Network Architecture

- Initialization of weights and biases for the additional hidden layer.
- Forward propagation through two hidden layers.
- Backward propagation and gradient calculations for both hidden layers.
- Parameter update mechanisms.

2. Data Preparation

- Loading the dataset using `fetch_openml`.
- Converting labels to integers.
- Normalizing feature values using `StandardScaler`.
- Splitting the data into training (70%) and testing (30%) sets with stratification.

3. Implementation Details

- Sigmoid Activation Function
- One-hot encoding of labels for multi-class classification.
- Loss function: Cross Entropy Loss
- LR = 0.1

  Batchsize = 64

  Hidden L1 = 128,

  Hidden L2 = 64

  Epoch = 50

4. Training Process

The training process of the neural network models involves iteratively refining the model's parameters to minimize the discrepancy between predicted outputs and actual labels. This process encompasses four fundamental steps: Forward Pass Computations, Loss Calculation, Backward Pass and Gradient Computation, and Parameter Updates. Both the custom two-hidden-layer implementation and the PyTorch-based ANN follow these steps.

4.1. Forward Pass Computations

Custom Implementation:

- **Input to Hidden Layers:**The model begins by taking the input features and passing them through the first hidden layer. This involves applying a linear transformation using the weights and biases of the first hidden layer. The result is then passed through the sigmoid activation function, which introduces non-linearity and produces the first set of activations.
- **Hidden to Hidden Layers:**The activations from the first hidden layer are then passed to the second hidden layer. Similar to the first layer, a linear transformation is applied using the weights and biases of the second hidden layer, followed by the sigmoid activation function. This produces the second set of activations.
- **Hidden to Output Layer:**Finally, the activations from the second hidden layer are passed to the output layer. Another linear transformation is applied using the output weights and biases. At this stage, the softmax function is applied to convert the outputs into probabilities, ensuring the predictions represent a distribution over the possible classes.

PyTorch Implementation:

- **Layer-wise Processing:** Similarly, the PyTorch model processes each input batch through successive linear layers (linear 1,2,3). After each linear transformation, the ReLU activation function (relu 1,2) is applied to introduce non-linearity.
- **Output Layer:** The final linear layer (fc3) outputs logits, which are then passed through the Softmax activation function to produce class probabilities.

4.2. Loss Calculation

Custom Implementation:

- **Cross-Entropy Loss:** The model computes the Cross-Entropy Loss by comparing the predicted probability distributions (a out) with the true one-hot

encoded labels (y onehot) This loss quantifies the difference between the predicted and actual distributions, guiding the optimization process.

PyTorch Implementation:

- **Built-in Loss Function:** The PyTorch model utilizes the `CrossEntropyLoss` function, which internally applies the Softmax activation to the output logits and computes the loss against the true class indices. This streamlined approach ensures numerical stability and efficient computation.

4.3. Backward Pass and Gradient Computation

Custom Implementation:

- **Backpropagation:**
  The model calculates how much each weight and bias contributes to the loss by performing backpropagation. Starting from the output layer, it computes the gradients of the loss with respect to the predictions and then propagates these gradients backward through the layers. The chain rule is used to calculate the gradients for every weight and bias in the network.
- **Gradient Components:**
  - **Output Layer:** Gradients are computed for the weights and biases in the output layer based on the difference between the predicted and actual values.
  - **Second Hidden Layer:** The error is propagated from the output layer back to the second hidden layer, and gradients for the weights and biases in this layer are calculated.
  - **First Hidden Layer:** The error is further propagated back to the first hidden layer, where the gradients for the weights and biases in this layer are computed.

This process ensures that all the parameters in the network are updated based on their contribution to the prediction error, helping the model learn effectively.

PyTorch Implementation:

- **Automatic Differentiation:** PyTorch leverages its `autograd` system to automatically compute gradients. After the forward pass and loss calculation, invoking `loss.backward()` computes the gradients of the loss with respect to all model parameters without manual intervention.

4.4. Parameter Updates

Custom Implementation:

- **Manual Gradient Descent:** The model updates its parameters by subtracting the product of the learning rate and the corresponding gradients from each weight and bias. This step iteratively adjusts the parameters to minimize the loss.

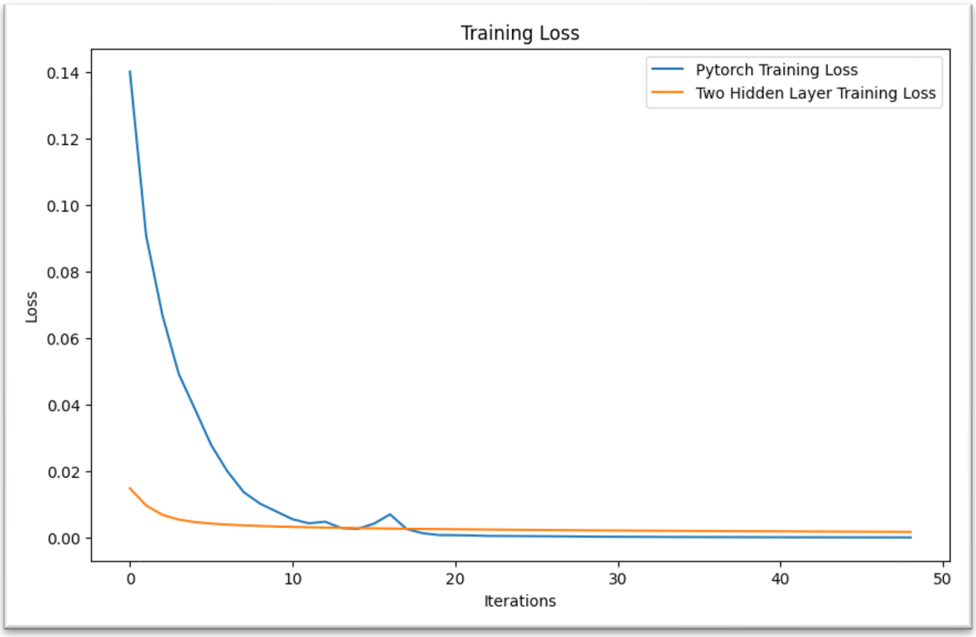
PyTorch Implementation:

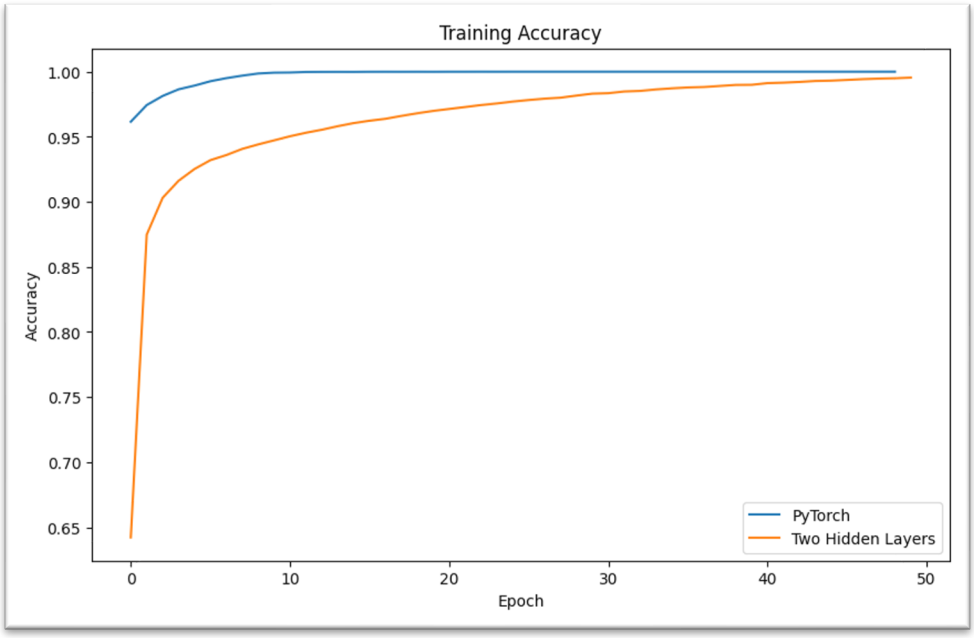
- **Optimizer Utilization:** PyTorch employs optimizers (e.g., SGD) to handle parameter updates. After computing gradients via loss.backward(), calling optimizer.step() updates all model parameters based on the gradients and the optimization algorithm's strategy.

4. Results

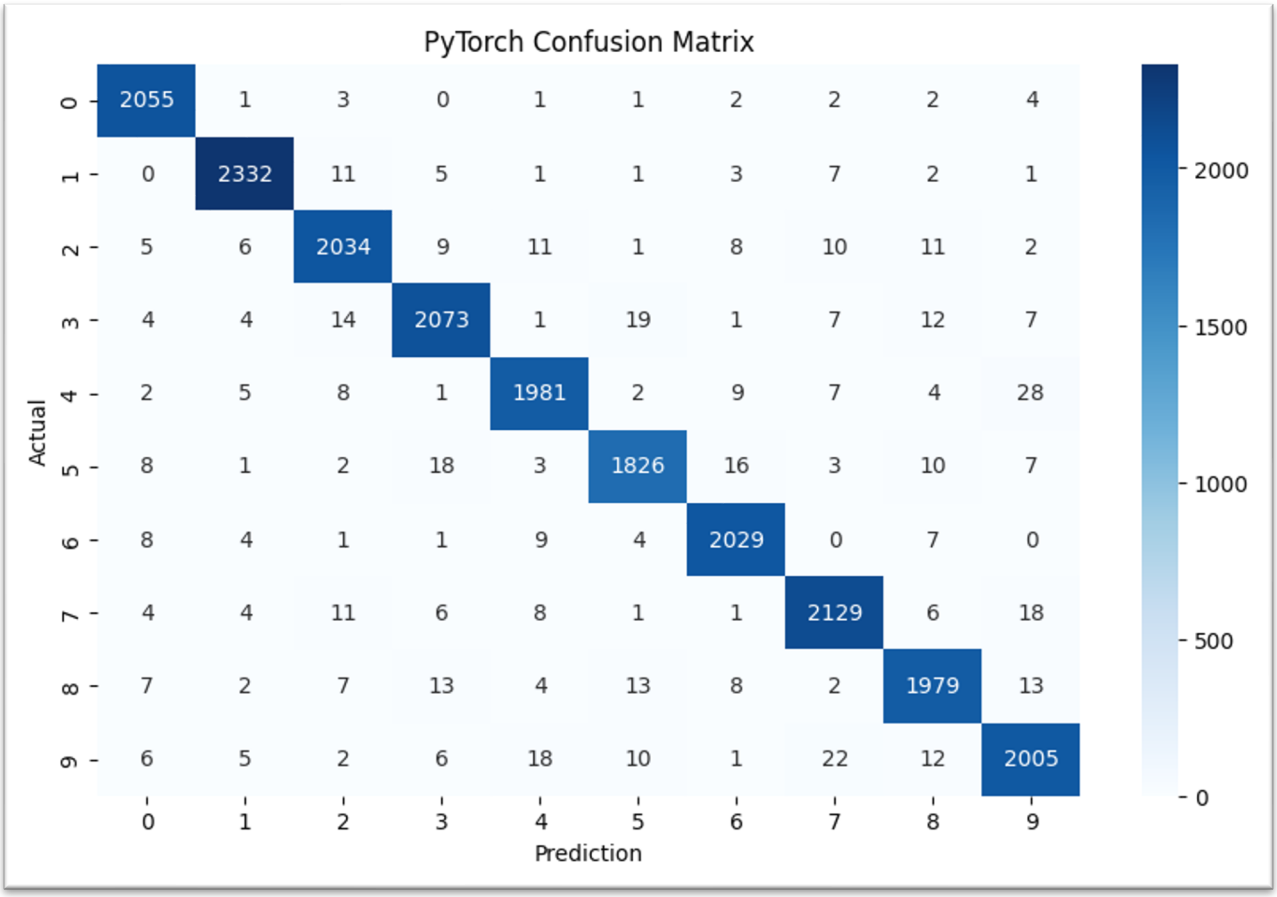
1. Performance Metrics

Model	Train Acc	Test Acc	Test AUC
Single Layer	0.9606	0.9401	
Two Layer	0.9955	0.9610	0.9983
Pytorch	0.981	0.9735	0.9958

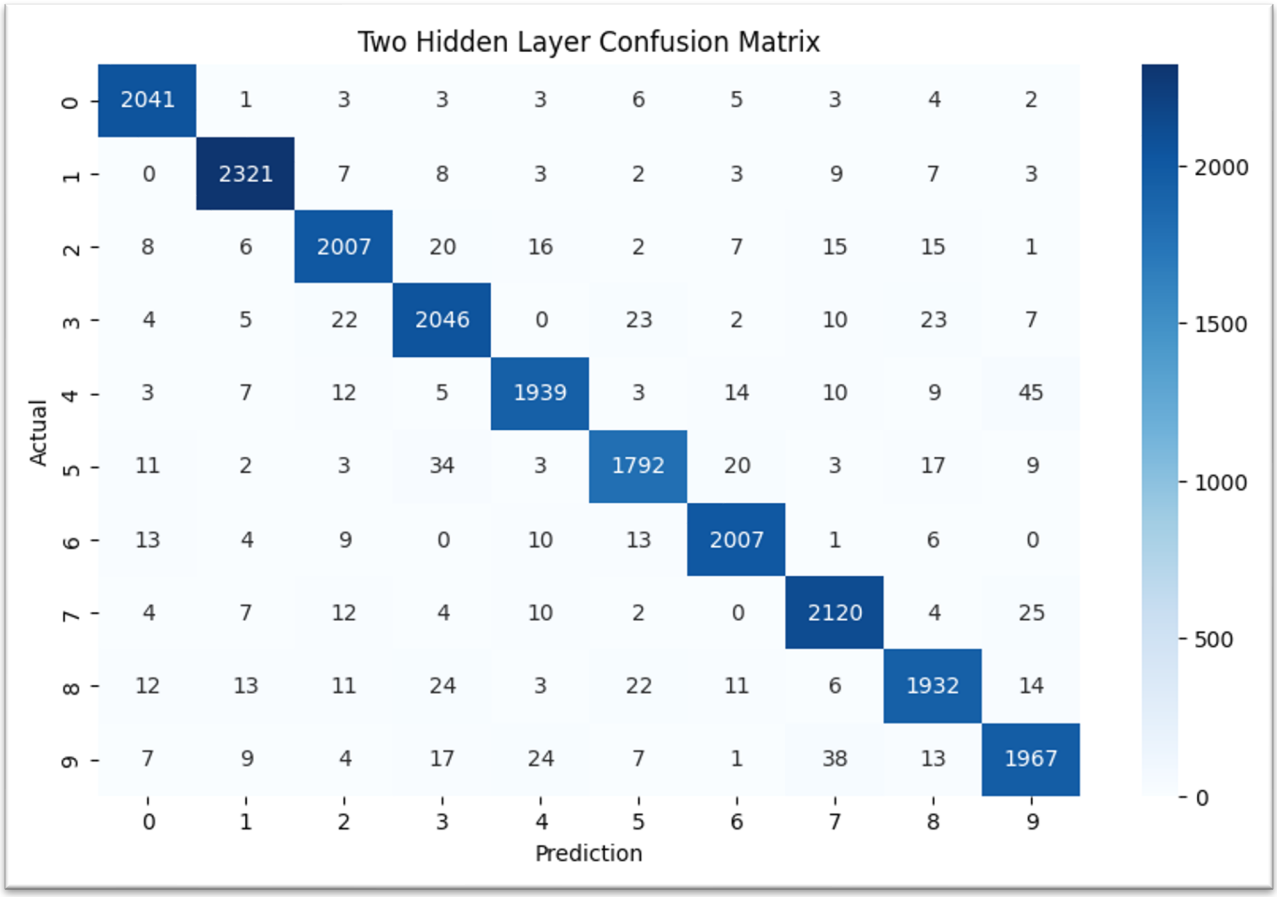




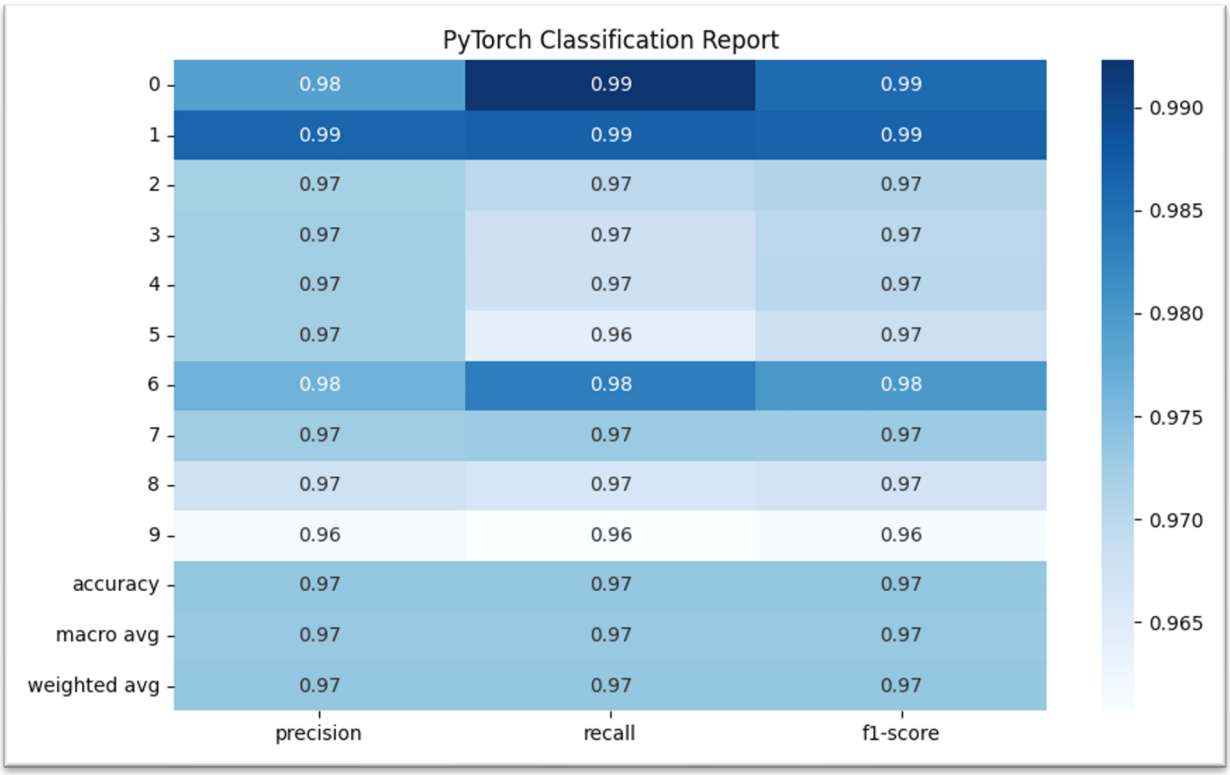
Confusion Matrix – Pytorch:

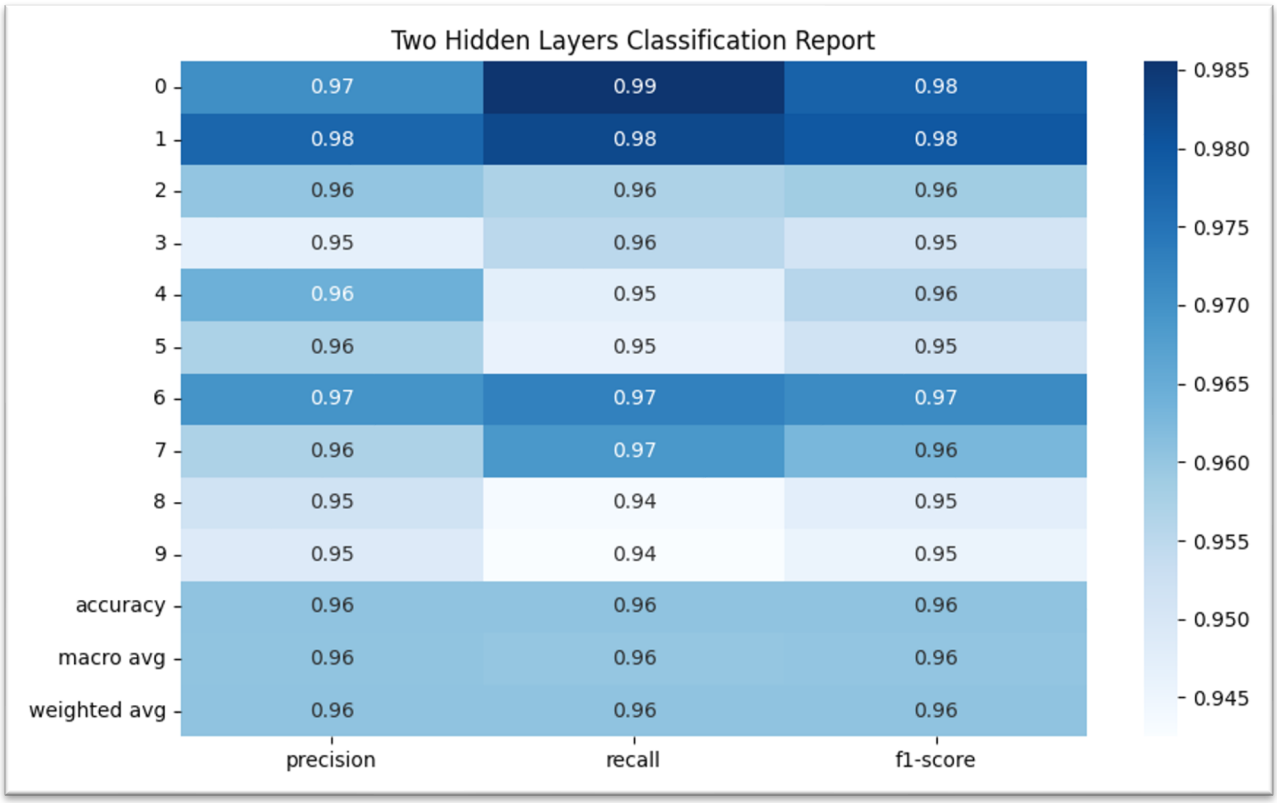


Confusion Matrix – Two Hidden Layers Custom Model:



Classification Reports:




Two Hidden Layers Classification Report
0
1
2
3
4
5
6
7
8
9
accuracy
macro avg
weighted avg
0.97 0.99 0.98
0.98 0.98 0.98
0.96 0.96 0.96
0.95 0.96 0.95
0.96 0.95 0.96
0.96 0.95 0.95
0.97 0.97 0.97
0.96 0.97 0.96
0.95 0.94 0.95
0.95 0.94 0.95
0.96 0.96 0.96
0.96 0.96 0.96
0.96 0.96 0.96
precision
recall
f1-score
0.985
0.980
0.975
0.970
0.965
0.960
0.955
0.950
0.945