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:** For each input batch, the model performs linear transformations by multiplying the input features with the weights of the first hidden layer (weight_h1) and adding the corresponding biases (bias_h1). The result is passed through the Sigmoid activation function to produce activations (a_h1).
- **Hidden to Hidden Layers:** The activations from the first hidden layer are then transformed using the weights (weight_h2) and biases (bias_h2) of the second hidden layer, followed by another Sigmoid activation to yield the second set of activations (a_h2).
- **Hidden to Output Layer:** Finally, the activations from the second hidden layer are multiplied by the output weights (weight_out) and biased (bias_out). The Softmax function is applied to the output layer to obtain probability distributions over the classes (a_out).

PyTorch Implementation:

- **Layer-wise Processing:** Similarly, the PyTorch model processes each input batch through successive linear layers (fc1, fc2, fc3). After each linear transformation, the ReLU activation function (relu1, relu2) 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 performs backpropagation by calculating the gradients of the loss with respect to each parameter (weights and biases) in the network. This involves computing the derivative of the loss with respect to the output activations, propagating these gradients backward through the network layers, and applying the chain rule to obtain gradients for each parameter.
- **Gradient Components:**
    - **Output Layer:** Computes gradients for weight_out and bias_out based on the error between predicted and actual outputs.
    - **Second Hidden Layer:** Derives gradients for weight_h2 and bias_h2 by propagating the error back from the output layer.
    - **First Hidden Layer:** Calculates gradients for weight_h1 and bias_h1 by further propagating the error back to the initial hidden layer.

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 (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	0.9646	0.9454	0.9938
Pytorch	0.981	0.9735	0.9958

5. References

- Scikit-learn Documentation: https://scikit-learn.org
- PyTorch Documentation: https://pytorch.org
- MNIST Dataset: http://yann.lecun.com/exdb/mnist/