Machine Learning

372.2.5214

Assignment 3 – Neural Network

Student: Ido Zimry

I.D: 205444219

Lecturer: Prof. Lior Rokah

Date: 15/01/2025

1. Chapter 11 of *"Machine Learning with PyTorch and Scikit-Learn"*

   Brief explanation of CH11:
   Focuses on implementing a **Perceptron Neural Network (NN)** from scratch:
   - Begins with an explanation of a single-layer perceptron, its structure, and how it uses **gradient descent** for learning.
   - Introduces the fundamental concept of training a model by minimizing a loss function.
   - Explains the architecture of a **Multi-Layer Perceptron (MLP)**, which includes an input layer, one or more hidden layers, and an output layer.
   - Discusses how additional layers enable the network to capture more complex patterns.
   - Covers preprocessing techniques, including **one-hot encoding** for categorical target labels, to prepare the data for training.
   - Describes how training data is passed through the network to compute the output.
   - Details the computation of the **loss** (error) using a loss function, which quantifies how far the predictions are from the true labels.
   - Introduces backpropagation, which computes the derivatives of the loss with respect to the model's weights.
   - Explains how these derivatives are used to update the weights using an optimization algorithm (e.g., gradient descent).
   - Illustrates the iterative process of training over multiple **epochs:**
     - Forward propagation to compute outputs.
     - Backward propagation to minimize the loss.
     - Weight and bias updates to improve the model's predictions.
   - Once trained, the network applies a **threshold function** on its outputs to determine the predicted class labels in a **one-hot encoded format**.

2. Modifying the original code, adding hidden layers explanation
   To add one more hidden layer to the NN, I will modify the 'NeuralNetMLP' class.

   With the same logic as already implemented, I will add to the class's attributes one more matrix of weights (number of neurons in layer X number of features) and an extra bias vector (number of neurons in layer).

   In the forward propagation method, the output vector from the firs layer will now be calculated with the second layer weights and bias, and will sent to the activation function (sigmoid):

$$z_1 = input \cdot W_1 + b_1$$
$$a_1 = \sigma(z_1)$$
$$z_2 = a_1 \cdot W_2 + b_2$$
$$a_2 = \sigma(z_2)$$
$$z_{out} = a_2 \cdot W_{out} + b_{out}$$
$$a_{out} = \sigma(z_{out}) = \hat{y}$$

   The backward method is where the gradient is calculated from the output layer, backward through the NN up to the input layer.
   The loss function is MSE:

$$L = \frac{1}{N}\sum_{i=1}^{N}(\hat{y_i} - y_i)^2$$

   The activation function is sigmoid:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

   Backpropagation for **output** layer will be:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial W} = \delta_{out} \cdot \frac{\partial z}{\partial W}$$

   Derivatives for output layer:

$$\left.\begin{array}{l} \dfrac{\partial L}{\partial \hat{y}} = \dfrac{2}{N}(\hat{y} - y) \\ \dfrac{\partial \hat{y}}{\partial z_{out}} = \hat{y}(1-\hat{y}) \end{array}\right. \rightarrow \ \delta_{out} = \frac{2}{N}(\hat{y} - y) \cdot \hat{y}(1-\hat{y})$$

$$\frac{\partial z_{out}}{\partial W_{out}} = a_2$$

$$\boldsymbol{\frac{\partial L}{\partial W_{out}} = \delta_{out} \cdot a_2} \quad , \qquad \boldsymbol{\frac{\partial L}{\partial b_{out}} = \Sigma\delta_{out}}$$

Backpropagation for the second hidden layer:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial W_2} = \delta_2 \cdot \frac{\partial z_2}{\partial W_2}$$

$$\frac{\partial z_2}{\partial W_2} = a_1$$

$$\frac{\partial a_2}{\partial z_2} = a_2(1 - a_2)$$

$$\frac{\partial L}{\partial a_2} = \delta_{out} \cdot W_{out}$$

$$\delta_2 = \delta_{out} \cdot W_{out} \cdot a_2(1 - a_2)$$

$$\boldsymbol{\frac{\partial L}{\partial W_2} = \delta_2 \cdot a_1} \quad , \qquad \boldsymbol{\frac{\partial L}{\partial b_2} = \Sigma\delta_2}$$

Same way of gradient calculation for the first hidden layer:

$$\boldsymbol{\frac{\partial L}{\partial W_1} = \delta_1 \cdot X} \quad , \qquad \boldsymbol{\frac{\partial L}{\partial b_1} = \Sigma\delta_1}$$

When the weights and biases will be updated using these gradient calculations:

$$W = W - \eta\frac{\partial L}{\partial W}$$

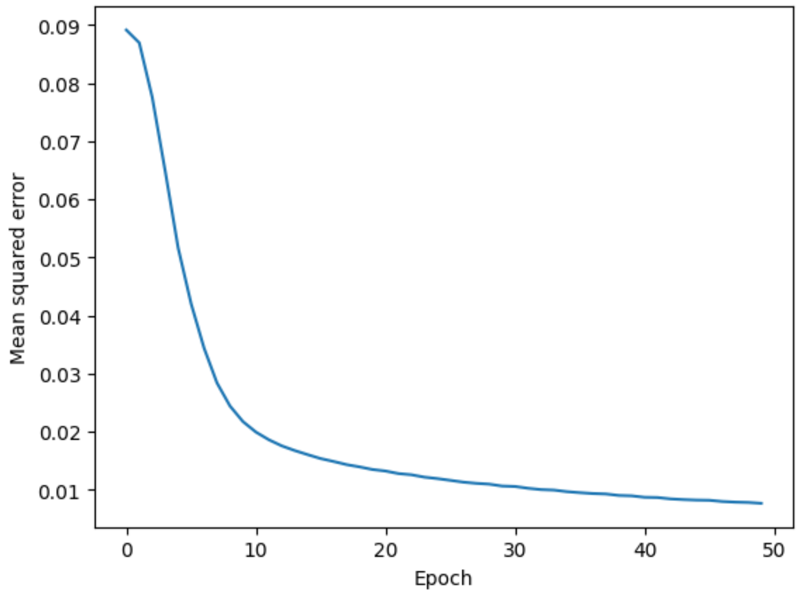
$$b = b - \eta\frac{\partial L}{\partial b}$$

This will ensure that the weight and biases are updated to minimize the loss function during the training. This process will repeat every epoch when training the network, with the given learning rate $\eta = 0.1$.

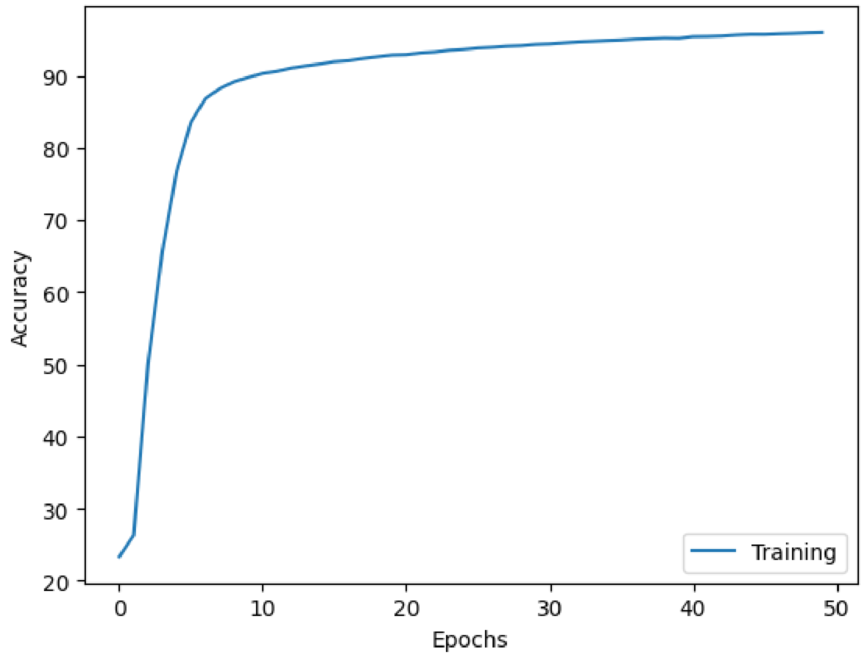
3. Evaluating the modified code from (2) with 2 hidden layers.
   The modified code with one extra hidden layer is executed with the same learning rate η, same number of epochs and 50 neurons in the second hidden layer.

   3.1. <u>Training process:</u>

   ➢ MSE Vs Epoch:

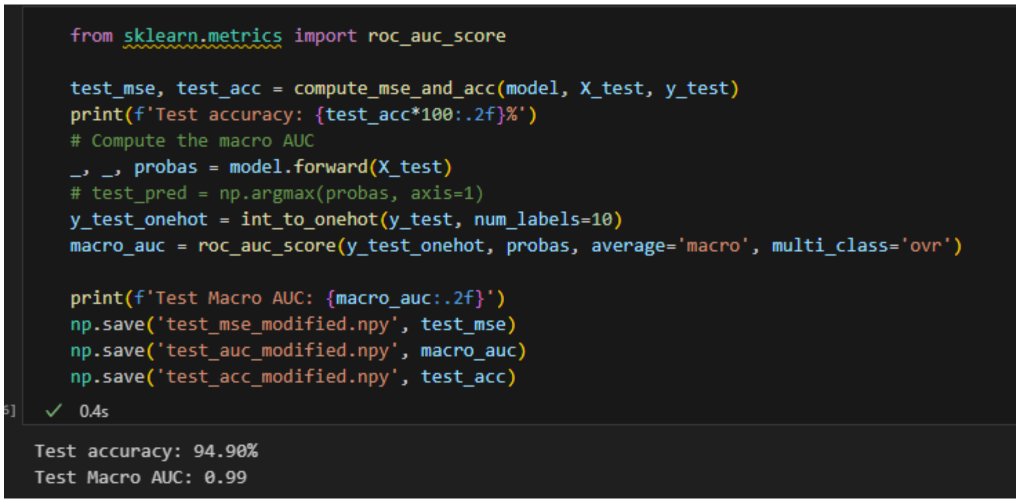


   ➢ Accuracy Vs Epoch:



3.2. Test accuracy and macro-AUC:

$$Test\ accuracy = 94.90\%$$
$$Test\ Macro - AUC = 0.99$$



```
from sklearn.metrics import roc_auc_score

test_mse, test_acc = compute_mse_and_acc(model, X_test, y_test)
print(f'Test accuracy: {test_acc*100:.2f}%')
# Compute the macro AUC
_, _, probas = model.forward(X_test)
# test_pred = np.argmax(probas, axis=1)
y_test_onehot = int_to_onehot(y_test, num_labels=10)
macro_auc = roc_auc_score(y_test_onehot, probas, average='macro', multi_class='ovr')

print(f'Test Macro AUC: {macro_auc:.2f}')
np.save('test_mse_modified.npy', test_mse)
np.save('test_auc_modified.npy', macro_auc)
np.save('test_acc_modified.npy', test_acc)
```

✓ 0.4s

```
Test accuracy: 94.90%
Test Macro AUC: 0.99
```

4. Implementing fully connected ANN using ‘pytorch’ package:
   The network that I implemented gets the input size, number and sizes of hidden layers and output layer size.
   First, splitting the data to 70/30 % when 70% of the data is used for the training process and the rest for testing.

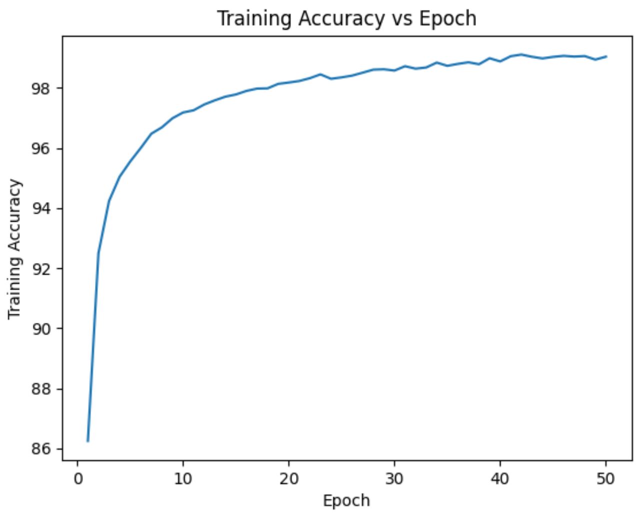
   - Activation function:
     All the hidden layers have the same activation function: ReLU.
     $$ReLU(x) = \max(0, x)$$

   - Loss function:
     The loss function in this network is Cross-Entropy:
     $$Loss = -\frac{1}{N}\sum_{i=1}^{N} \log\left(P(y_i|x_i)\right)$$

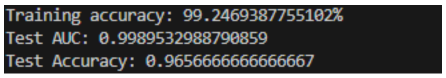
     As P is the predicted probability of the correct class calculated by ‘Softmax’ – which convert the raw logits to probabilities.

   - Training process:
     i. The input data is passed through the network to compute the predictions.
     ii. The predictions are compared with the true labels and using cross-entropy the loss is calculated.
     iii. Backpropagation – gradients of the loss are computed.
     iv. Weights and biases update using the ‘Adam’ optimizer to minimize the loss.

- Network evaluation:
  At first, running this network with the same epochs and hidden layers as the modified NN from (2) and with $\eta = 0.01$:

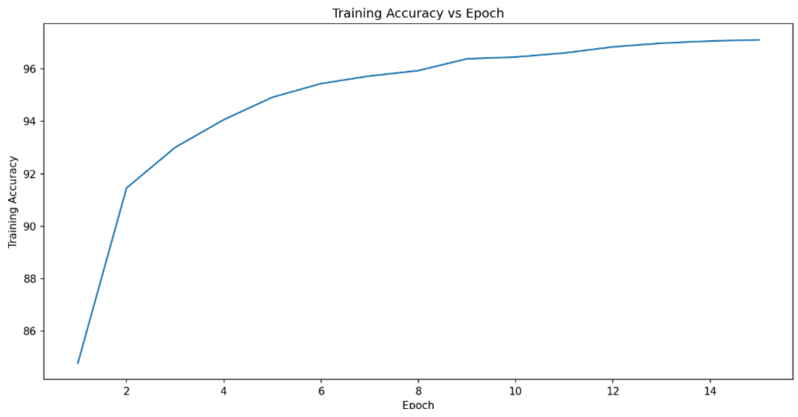


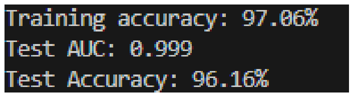
From the graph it can be seen that after less than 20 epochs the training accuracy is above 96, which can imply that the number of epochs can be significantly smaller.



```
Training accuracy: 99.2469387755102%
Test AUC: 0.9989532988790859
Test Accuracy: 0.9656666666666667
```

The training accuracy is 99.2 and the test accuracy is 96.5 which means that the model is well fitted and generalized.
To reduce the computational time, I can reduce the size of the hidden layer and the number of epochs, after tuning:
$epochs = 15, hidden1 = 50,\ hidden2 = 25:$





```
Training accuracy: 97.06%
Test AUC: 0.999
Test Accuracy: 96.16%
```

As the training accuracy and the test accuracy are high, and there is a small gap between them, it suggests that the model is still well fitted and generalized.

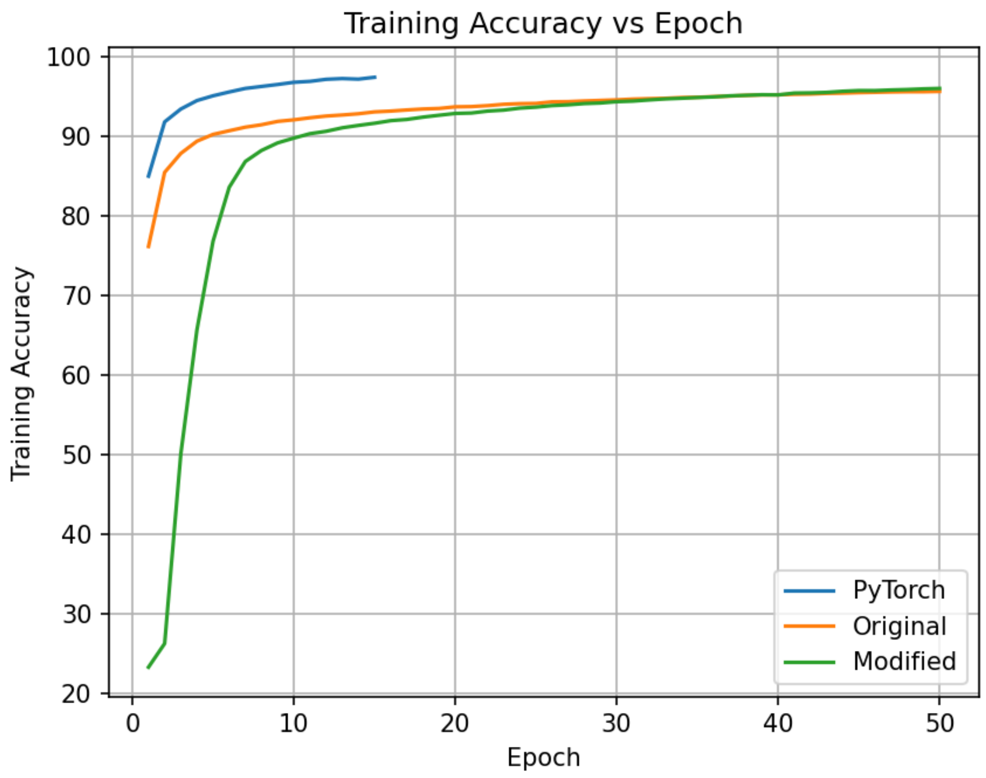
5. Models Comparison:

   The three models:

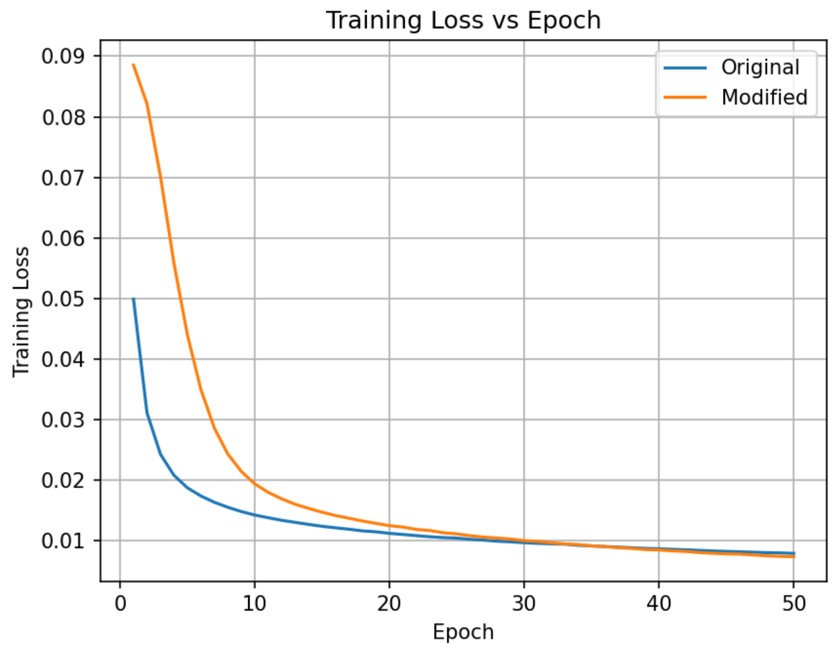
   i. Original NN from ch11 – one hidden layer
   ii. Modified NN from ch11 – two hidden layers
   iii. NN using pytorch – two hidden layers

   As mentioned above, the two NN from ch11 uses sigmoid activation function and MSE loss function, while the NN implemented with pytorch uses ReLU activation function and Cross-Entropy loss function.

   - Training accuracy Vs Epochs:



- Training loss Vs Epochs (ch11 models):



- Models comparison:

Model	Epochs	Test Accuracy (%)	Test AUC
PyTorch	15	96.157	0.999
Original	50	94.54	0.993
Modified	50	94.895	0.994

6. Conclusion:

In this assignment, I began by using the perceptron NN with one hidden layer, from the example in Chapter 11. I then modified this NN by adding a second hidden layer while keeping key parameters constant, such as using sigmoid activation functions, the mean squared error (MSE) loss function, 50 epochs, and the same learning rate. The comparison between the original and modified models shows that the modified NN achieved **slightly** better results, with higher training accuracy, lower training loss, and marginally improved test accuracy. However, the performance gain was not substantial.

Next, I implemented a new NN using PyTorch. This NN differed from the previous implementations in several key ways: it used the ReLU activation function, cross-entropy loss function, a smaller second hidden layer of size 25, and was trained for only 15 epochs—significantly fewer than the 50 epochs used in the other models. Despite the reduced training time and smaller hidden layer, the PyTorch NN outperformed both the original and modified models. It achieved the highest test accuracy (96.157%) and the highest AUC (0.999) among all three models. This demonstrates the benefits of using ReLU and cross-entropy loss, along with a more efficient architecture, in improving both performance and training efficiency.

Overall, while the modifications to the original NN provided slight improvements, the PyTorch implementation demonstrated superior performance and efficiency, highlighting the importance of activation functions, loss functions, and architectural design in neural network optimization.