Simple MLP Framework for Classification (March 2024)

David A. Arungbemi
191203012
Department of Computer Engineering
Nile University of Nigeria
Abuja, Nigeria

***Abstract*—This report presents the methodology and outcomes of developing a simple Multilayer Perceptron (MLP) framework for classification tasks using only the NumPy library. The objective was to construct a fundamental neural network architecture capable of classification. The process involved designing and implementing the core components of an MLP, including feedforward propagation, backpropagation, parameter initialization, activation functions, and gradient descent optimization. Additionally, the framework was evaluated through experimentation with a real-world dataset to assess its performance in classification scenarios. The results demonstrate the efficacy of the developed framework in achieving accurate predictions while showcasing its simplicity and efficiency.**

***Index Terms*—Perceptron, Multilayer Perceptron, Backpropagation, Partial Derivatives, Gradient Descent, Softmax, ReLU, Sigmoid, Mean Squared Error, Cross Entropy, Weights, Biases, Matrix Multiplication, Hadamard Product**

I. Introduction

Artificial intelligence and machine learning have advanced immensely in recent years and have transformed various domains. Among the machine learning techniques, Multilayer Perceptrons (MLPs) have gained significant attention due to their ability to learn complex patterns from data. MLPs are powerful tools for regression and classification tasks, with interconnected layers of neurons.

However, despite the availability of sophisticated deep learning frameworks, there remains a need for simpler, more transparent implementations that offer a deeper understanding of the underlying principles of neural networks. This report addresses this need by detailing the development of a basic MLP framework designed for classification tasks, utilizing only the NumPy library.

The primary objective of this work is to construct a foundational neural network architecture that encapsulates the essential components required for effective classification. This includes the implementation of feedforward propagation, backpropagation, parameter initialization, activation functions, and gradient descent optimization – all integral aspects of neural network training and inference.

To evaluate the effectiveness and adaptability of the framework, we conduct testing using a real-world dataset. The goal is to demonstrate the framework's ability to classify inputs accurately.

This report aims to contribute to facilitating a deeper understanding of neural network fundamentals and providing a solid foundation in the field of machine learning.

II. Neural Networks as Functions

Neural networks are essentially functions because they operate on the fundamental principle of transforming input data into output data through a series of mathematical operations [1]. At its core, a neural network consists of interconnected nodes, called neurons, organized into layers [2].

Each neuron performs a simple computation: it takes in the weighted sum of its inputs, adds a bias term, and applies an activation function to produce an output [2]. This process can be represented mathematically as follows:

$$y = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right)$$

Where w_i is the weight, b is the bias, x is the input and y is the output.

The choice of activation function introduces non-linearity into the network, allowing it to capture complex relationships in the data [2]. Common activation functions include sigmoid, tanh, ReLU (Rectified Linear Unit), and softmax.

By arranging neurons into layers and connecting them in a specific topology, neural networks can learn to approximate complex functions that map inputs to outputs. During training, the network adjusts its weights and biases iteratively using optimization algorithms such as gradient descent. The objective is to minimize a loss function, which measures the difference between the network's predictions and the actual target values [3].

This iterative optimization process enables neural networks to learn from data and generalize their knowledge to make predictions on unseen examples. As a result, neural networks can be thought of as highly adaptable mathematical functions that can model intricate patterns and relationships in various types of data, making them invaluable tools in machine learning and artificial intelligence applications.

III. MLP Framework: Activation and Loss Function

A. *Loss Function*

The Mean Squared Error measures the average squared difference between the predicted and actual values in regression problems [4].

Formula:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where: N is the number of samples, y_i is the actual target for sample i, $\hat{y}_i$ is the predicted target for sample i.

Cross Entropy is a measure of dissimilarity between the true distribution of the data and the predicted distribution. In the context of classification, it quantifies the difference between the predicted probability distribution and the true distribution of the labels [5].

Formula:

$$H(P,Q) = -\sum_{i} P(i)\log(Q(i))$$

Where: $H(P,Q)$ represents the cross entropy between probability distributions P, and Q $P(i)$ is the probability of event i according to distribution P, $Q(i)$ is the probability of event i according to distribution Q.

B. Activation Function

The ReLU (Rectified Linear Unit) activation function is a simple yet powerful non-linear function commonly used in deep learning. It introduces non-linearity by outputting the input directly if it is positive, and zero otherwise. ReLU has the advantage of being computationally efficient and preventing the vanishing gradient problem during training [6].

Formula:

$$f(x) = \max(0, x)$$

The Softmax activation function is widely used as the output layer activation in multi-class classification problems. It converts the raw output scores (logits) into a probability distribution over multiple classes. Softmax ensures that the output probabilities sum up to 1, making it suitable for interpreting the model's confidence in each class prediction [6].

Formula for j-th output:

$$f(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

Where: x_j is the input to the j-th neuron, K is the total number of output neurons.

The Sigmoid activation function is commonly used in binary classification tasks. It squashes the input values into the range (0, 1), interpreting them as probabilities. Sigmoid is advantageous for its smoothness and its ability to produce well-calibrated probability estimates, making it suitable for binary decision-making tasks [6].

Formula:

$$f(x) = \frac{1}{1 + e^{-x}}$$

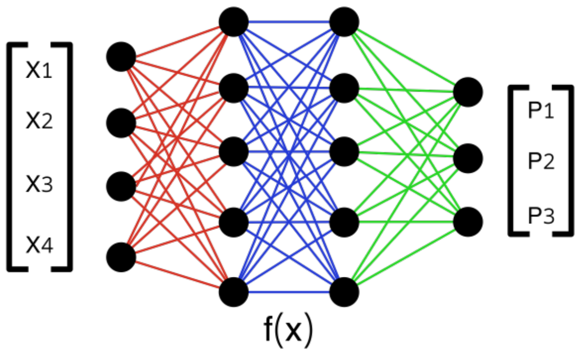


Fig. 1: MLP topology for learning the Iris Dataset. The column matrix $\mathbf{X}$ represents the 4 input features of the dataset. The column matrix $\mathbf{P}$ represents the probabilities of the 3 possible classes (flower types).

IV. MLP Framework: Iris Dataset Model Architecture

To demonstrate the functionality of the MLP framework, the Iris dataset is employed as an example. This dataset comprises 4 features, namely 'sepal_length', 'sepal_width', 'petal_length', and 'petal_width', and a target variable that represents the species of the iris flower. The target variable has 3 possible classes, namely 'Iris-setosa', 'Iris-versicolor', and 'Iris-virginica'.

The MLP consists of 4 input perceptrons (input features), 2 hidden layers with 5 perceptrons with a ReLU activation function each and 3 output perceptrons (targets) with a softmax activation function. Figure 1 is an illustration of the MLP architecture. Listing 1 shows the parameters of our MLP model in Python.

Listing 1: Instantiation of a Multilayer Perceptron (MLP) neural network with specified parameters

```
nn = MLP(
    layers = (4,5,5,3),
    lr = 1e-3,
    activation = 'relu',
    l_o_activation = 'softmax',
    cost = 'cross_entropy',
    random_seed = 42,
    weight_range = (-0.1, 0.1)
)
```

A. Weight and Bias Generation

The weight connections (red, blue and green) between the layers in Figure 1 can be represented with matrices. Matrices are 2-dimensional arrays of data consisting of rows and columns. In NN, the size of weight matrices representing the connection between layers is illustrated in Figure 2. The number of columns is equivalent to the number of perceptrons in the previous layer while the number of rows is equivalent to the number of perceptrons in the next layer. Listing 2 shows the Python implementation for generating our weights

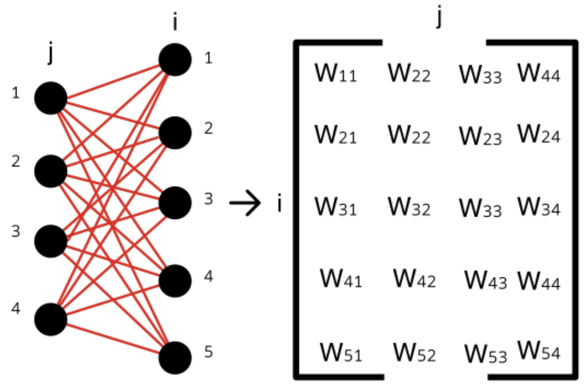


Fig. 2: 5x4 weight matrix representation of MLP connection from Input Layer to first hidden layer.

and biases. Each neuron also has a bias, which provides the neuron with more freedom when providing output. Bias in each layer can presented by a column matrix (1xn), with each value corresponding to a particular perceptron.

Listing 2: Utilising python list comprehension to generate weight and bias matrices based on perceptron count of the present and previous layer. `min_w` and `max_w` represent the range for generating weights. `size` represents the row and column length of the matrix. Weights are generated at random following a uniform distribution.

```
self.weights = [
    np.random.uniform(
        min_w, max_w,
        size=(layers[i],layers[i - 1])
    )
    for i in range(1,len(layers))
]

self.bias = [
    np.random.uniform(
        min_w, max_w, size=(layers[i], 1)
    )
    for i in range(1,len(layers))
]
```

B. Forward Propagation

In forward propagation, the inputs are passed through the MLP which transforms it to the desired output form which is basically what a function does. In MLPs, it is a series of matrix multiplication from one layer to the next, however after each multiplication, a bias is added. Listing 3 shows the Python implementation of forward propagation.

Listing 3: Implementation of forward propagation in python. Note `layer_history` stores the output (before and after activation) after each layer.

```
def forward(self, X: np.ndarray):
    assert isinstance(X, np.ndarray),\
    `Should be a numpy array`
    assert X.ndim == 2,\
    `X should be 2 dimension`

    Z = X.T.copy()
    self.layer_history = [[None, Z]]
    for i in range(len(self.weights)-1):\
    #to multiply through weights\
    in each hidden layer

        layer_store = []
        W = self.weights[i]

        Z = np.dot(W, Z) + self.bias[i]\
        #matrix multiplication + bias

        layer_store.append(Z)
        Z = Activation(self.activation,\
            Z, False)\
        #applying activation function

        layer_store.append(Z)

        self.layer_history.append\
        (layer_store) \
        #to store outputs in each\
        layer later for backpropagation

    # output layer
    layer_store = []
    W = self.weights[-1]
    Z = np.dot(W, Z) + self.bias[-1]
    layer_store.append(Z)
    Z = Activation(self.l_o_activation,\
        Z, False)

    layer_store.append(Z)
    self.layer_history.append(layer_store)

    return Z
```

1) Input Layer and Hidden Layer 1: Let X be the 4×1 node input matrix of features:

$$X = \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{bmatrix}$$

Let W_1 be the 5×4 weight matrix between the input layer and hidden layer 1:

$$W_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{bmatrix}$$

And let B_1 be the 5×1 bias matrix:

$$B_1 = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \\ b_{51} \end{bmatrix}$$

The resulting matrix Z_1 will be a 5×1 matrix:

$$Z_1 = W_1 \cdot X + B_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{bmatrix} \cdot \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \\ b_{51} \end{bmatrix}$$

Where Z_1 is the output of the first hidden layer.

2) Hidden Layer 1 and Hidden Layer 2: Here, similar logic follows. Let W_2 be the 5x5 weight matrix between hidden layer 1 and hidden layer 2:

$$W_2 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix}$$

The resulting matrix Z_2 will be a 5×1 matrix:

$$Z_2 = W_2 \cdot Z_1 + B_2 =$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix} \cdot \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}$$

Where Z_2 is the output of the second hidden layer.

3) Hidden Layer 2 and Output Layer: Let W_3 be the 3x5 weight matrix between hidden layer 2 and the output layer:

$$W_3 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{bmatrix}$$

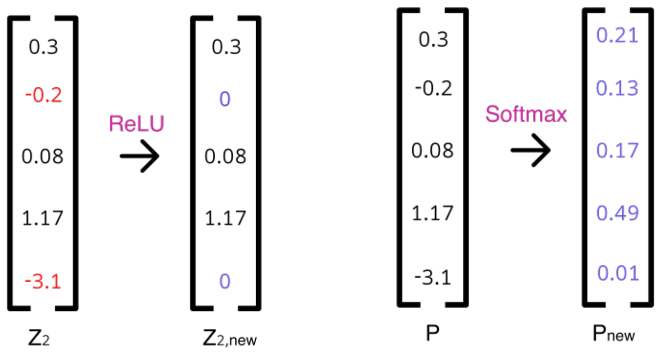
The final matrix P will be a 3×1 matrix:

$$P = W_3 \cdot Z_2 + B_3 =$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{bmatrix} \cdot \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Where P is the final layer output and result of the MLP.

4) ReLU and Softmax: The activation functions are applied after each layer, they do not affect the shape of our output. The ReLU function is applied after each hidden layer and allows the MLP to learn non-linear patterns in the Iris dataset. The softmax function is applied after the output layer and converts the raw values to the probabilities of each flower type. Figure 3 illustrates the effect of ReLU and softmax on the layer outputs.



(a) ReLU (b) Softmax

Fig. 3: Effect of ReLU and Softmax on outputs. ReLU turns all negatives to 0. Softmax transforms raws to probability distributions summing up to 1.

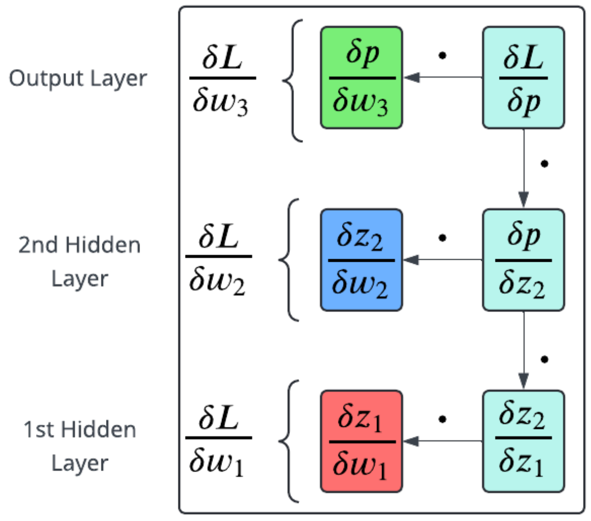

Output Layer
2nd Hidden Layer
1st Hidden Layer
$\frac{\delta L}{\delta w_3}$
$\frac{\delta p}{\delta w_3}$
$\frac{\delta L}{\delta p}$
$\frac{\delta L}{\delta w_2}$
$\frac{\delta z_2}{\delta w_2}$
$\frac{\delta p}{\delta z_2}$
$\frac{\delta L}{\delta w_1}$
$\frac{\delta z_1}{\delta w_1}$
$\frac{\delta z_2}{\delta z_1}$

Fig. 4: Backpropagation: Deriving the loss function gradient to each layer weight of the MLP using chain rule approach. Note: The dot symbols represent the dot product.

C. Backpropagation

In this step, the loss (cross-entropy) gradient of each weight is calculated. Gradients can be thought of as derivatives. This technique starts backwards from the output layer, calculates the derivative of the loss function to the weights and repeats the process in the other layers, hence we utilise the chain-rule differentiation technique as shown in Figure 4.

Backpropagation for the MLP model involves calculating the derivative of the cross-entropy function to each of the weights in our neural network and we achieve this using the chain rule. The process starts from the output layer back to the first hidden layer.

1) Output Layer: The derivative of the loss function to the output p, weights w_3, bias b_3, and input z_2, is calculated:

$$\frac{\delta L}{\delta p} = s - d = \begin{bmatrix} s_1 - d_1 \\ s_2 - d_2 \\ s_3 - d_3 \end{bmatrix}$$

$\frac{\delta L}{\delta p}$ is the cross entropy derivative to the predicted probability, p. s is the predicted probability distribution of the output layer result (after applying softmax). d is the true distribution to be achieved. $s - d$ gives the distance between the desired and predicted probabilities.

$$\frac{\delta L}{\delta w_3} = \frac{\delta L}{\delta p} \cdot \frac{\delta p}{\delta w_3}(w_3 \cdot z_2 + b_3)$$

$$= (s - d) \cdot z_2$$

z_2 is the previous layer output (input to output layer).

$$\frac{\delta L}{\delta b_3} = \frac{\delta L}{\delta p} \cdot \frac{\delta p}{\delta b_3}(w_3 \cdot z_2 + b_3)$$

$$= (s - d)$$

$$\frac{\delta L}{\delta z_2} = \frac{\delta L}{\delta p} \cdot \frac{\delta p}{\delta z_2}(w_3 \cdot z_2 + b_3)$$

$$= w_3$$

Weights and biases are updated as follows:

$$w_3 = w_3 - lr \cdot \frac{\delta L}{\delta w_3}$$

$$b_3 = b_3 - lr \cdot \frac{\delta L}{\delta b_3}$$

Where lr is the learning rate (< 1).

2) Hidden Layer 2: The derivative of the loss function to the weights, bias and input is calculated:

Weights

$$\frac{\delta L}{\delta w_2} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta w_2}(w_2 \cdot z_1 + b_2)$$

Bias

$$\frac{\delta L}{\delta b_2} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta b_2}(w_2 \cdot z_1 + b_2)$$

Input

$$\frac{\delta L}{\delta z_1} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta z_1}(w_2 \cdot z_1 + b_2)$$

Weights and biases are updated as follows:

$$w_2 = w_2 - lr \cdot \frac{\delta L}{\delta w_2}$$

$$b_2 = b_2 - lr \cdot \frac{\delta L}{\delta b_2}$$

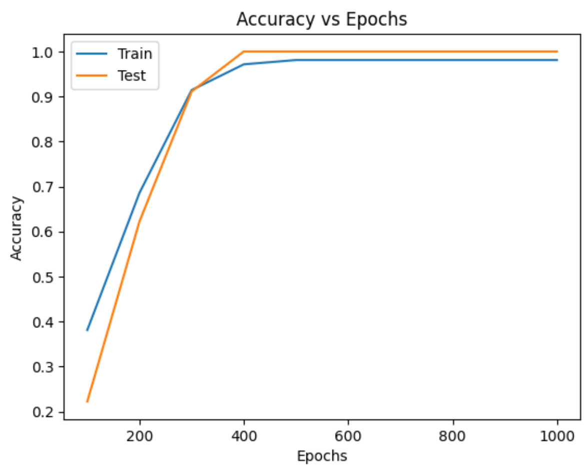


Fig. 5: Accuracy of the MLP model on Iris dataset over a range of epochs. Note: the shuffle for data used a random seed of 42.

3) Hidden Layer 1: The derivative of the loss function to the weights and bias is calculated:

Weights

$$\frac{\delta L}{\delta w_1} = \frac{\delta L}{\delta z_1} \cdot \frac{\delta z_1}{\delta w_1}(w_1 \cdot x + b_1)$$

Bias

$$\frac{\delta L}{\delta b_1} = \frac{\delta L}{\delta z_1} \cdot \frac{\delta z_1}{\delta b_1}(w_1 \cdot x + b_1)$$

Weights and biases are updated as follows:

$$w_1 = w_1 - lr \cdot \frac{\delta L}{\delta w_1}$$

$$b_1 = b_1 - lr \cdot \frac{\delta L}{\delta b_1}$$

The process is repeated over all the samples of the dataset in each epoch (iteration) till the desired loss/accuracy is achieved.

See Appendix A for backpropagation Python implementation.

V. MLP Framework: Model Results

As illustrated in Figure 5, an increase in the number of epochs leads to improved model accuracy when assessed on the Iris dataset. Initially, the training accuracy consistently outperforms the testing accuracy, which aligns with the notion that the model learns from the training data. However, at around 300 epochs, both training and testing accuracies converge to approximate equality. Continuing to increase the epochs, the testing accuracy surpasses the training accuracy. Beyond approximately 400 epochs, both accuracies stabilize until the final 1000 epochs. Ultimately, the training accuracy reached 97.1 %, while the testing accuracy achieved 100 %.

After conducting several tests, it was determined that the fluctuation in model performance was attributed to the random seed used for shuffling the Iris data, specifically seed 42. By switching to an alternative seed, such as 32, the training accuracy consistently remained higher than the testing accuracy throughout the 1000 epochs, as depicted in Figure 6.

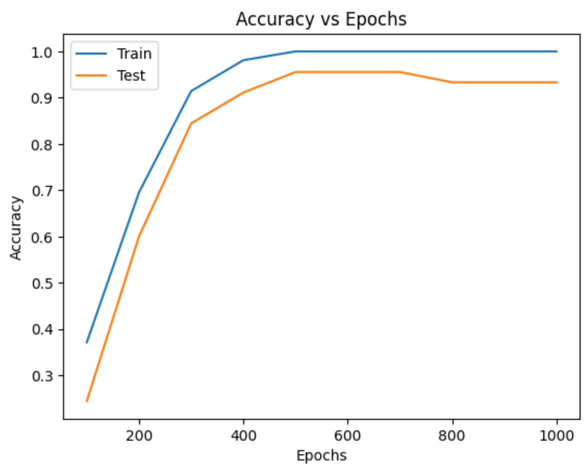


Fig. 6: Accuracy of the MLP model on Iris dataset over a range of epochs. Note: the shuffle for data used a random seed of 32.

Notably, around the 700-epoch mark, there was a decrease in test accuracy while the training accuracy remained relatively unchanged, indicating overfitting. Ultimately, the training accuracy reached 98.1 %, while the testing accuracy achieved 91.11 %.

After conducting several tests, it was determined that the fluctuation in model performance was attributed to the random seed used for shuffling the Iris data, specifically seed 42. By switching to an alternative seed, such as 32, the training accuracy consistently remained higher than the testing accuracy throughout the 1000 epochs, as depicted in Figure 6. Notably, around the 700-epoch mark, there was a decrease in test accuracy while the training accuracy remained relatively unchanged, indicating overfitting. Ultimately, the training accuracy reached 98.1 %, while the testing accuracy achieved 91.11 %.

The fluctuations observed in the train-test accuracy relationship depicted in Figure 5 may have stemmed from the choice of a random seed, specifically seed 42, used for shuffling the data. The test dataset could have exhibited lower variability compared to the training dataset, potentially leading to the model performing exceptionally well on the test data due to its reduced variation. When data is shuffled with a seed, that seed may create a scenario where the test dataset contains less diverse samples compared to the training dataset. Consequently, during training, the model encounters a wider range of patterns and features, learning to adapt to this variability. However, when evaluating the model's performance on the test data, the reduced variation can lead to favourable outcomes. This scenario may create a false impression of high generalization performance.

VI. Conclusion

In conclusion, the development of the simple Multilayer Perceptron (MLP) framework using only the NumPy library has been successfully achieved, as outlined in this report. Through meticulous design and implementation, we have constructed a fundamental neural network architecture tailored for classification tasks. The methodology encompassed the creation of essential components including feedforward propagation, backpropagation, parameter initialization, activation functions, and gradient descent optimization.

The evaluation of the framework against real-world datasets has provided valuable insights into its performance in classification scenarios. Results on the Iris dataset show the model can achieve $> 90\%$ accuracy on both train and test data which demonstrates the capability of the developed framework in generating accurate predictions.

Recognizing the limitations encountered during the development and evaluation stages highlights the importance of refining the sampling methodology. As observed in Figure 5, the results were influenced by the sampling method, particularly the utilisation of random state sampling. For datasets like the Iris dataset with a limited number of samples, the adoption of stratified sampling would be more conducive to accurate representation. Furthermore, the absence of early stopping implementation to select optimal weights and the inefficiency of updating weights sample by sample for the entire dataset in each epoch contributed to a notably slow training process. Implementing stratified sampling, incorporating early stopping mechanisms, optimizing weight update and more are essential steps towards improving the framework's performance, ensuring it remains robust and performs accurately in the classification of tasks across various datasets.

VII. Appendix A

Listing 4: Backpropagation using `backward` and `chain_rule` function. Note: `backward` runs once and represents the output layer. `chain_rule` represents the hidden layers.

```
def backward(self, y_true: np.ndarray):
        weights_ = self.weights.copy()
        #before activation
        Z = self.layer_history[-1][0]\
        #after activation
        y_pred = self.layer_history[-1][1]\

        dC_dZ = None
        if self.l_o_activation != 'softmax':
            dC_dA = Loss_Fn(self.cost,\
                y_true, y_pred, True)

            dA_dZ = Activation(\
                self.l_o_activation,\
                Z, True)
            dC_dZ = dC_dA * dA_dZ

        else:
            true_distr = np.array(\
            [np.array([0])\
            if i != y_true[0][0]\
            else np.array([1])\
            for i in range(Z.shape[0])])
```

```
                dC_dZ = Activation(\
                    self.l_o_activation,\
                    Z, False) - true_distr

            dZ_dW = self.layer_history[-2][1]
            dC_dW = np.dot(dC_dZ, dZ_dW.T)
            dC_dB = dC_dZ

             #update weight at layer
            self.weights[-1] -= dC_dW * self.lr
            #update bias at layer
            self.bias[-1] -= dC_dB  * self.lr

            if len(weights_) != 1:
                self.chain_rule(\
                    np.dot(dC_dZ.T,\
                    weights_[-1]),\
                    weights_)


    def chain_rule(self, dC_dA, weights_):
        self.layer_history.pop()
        weights_.pop();
        for _ in range(len(weights_)):
            Z = self.layer_history[-1][0]

            dA_dZ = Activation(\
                        self.activation,\
                        Z, True
                        )
            dZ_dW = self.layer_history[-2][1]

            dC_dZ = dC_dA.T * dA_dZ
            dC_dW = np.dot(dC_dZ, dZ_dW.T)
            dC_dB = dC_dZ

            #update weight at layer
            self.weights[len(weights_) - 1]\
                -= dC_dW * self.lr

            #update bias at layer
            self.bias[len(weights_) - 1]\
                -= dC_dB  * self.lr

            dC_dA = np.dot(dC_dZ.T,\
                weights_[-1])

            self.layer_history.pop()
            weights_.pop();
```

References

[1] Wikipedia Contributors, "Universal approximation theorem," Wikipedia, Jan. 20, 2020. Available: https://en.wikipedia.org/wiki/Universal_approximation_theorem

[2] "Neural network (machine learning)," Wikipedia, Feb. 18, 2024. Available: https://en.wikipedia.org/wiki/Neural_network_(machine_learning)

[3] A. Roy, "An Introduction to Gradient Descent and Backpropagation," Medium, Jun. 14, 2020. Available: https://towardsdatascience.com/an-introduction-to-gradient-descent-and-backpropagation-81648bdb19b2

[4] Wikipedia Contributors, "Mean squared error," Wikipedia, Mar. 30, 2019. Available: https://en.wikipedia.org/wiki/Mean_squared_error

[5] "Cross-entropy," Wikipedia, Mar. 30, 2024. Available: https://en.wikipedia.org/wiki/Cross-entropy#:~:text=In%20information%20theory%2C%20the%20cross. [Accessed: Apr. 02, 2024]

[6] Wikipedia Contributors, "Activation function," Wikipedia, Nov. 30, 2018. Available: https://en.wikipedia.org/wiki/Activation_function