University of Siena

Department of Information Engineering and Mathematics

Multilayer Perceptron

Kevin Gagliano

Pietro Pianigiani

Fabrizio Benvenuti

Supervisor:

Prof. Marco Gori

Academic Year 2023-2024

Contents

Chapter 1

Introduction

1.1 Background

The concept of the multilayer perceptron (MLP) holds significant importance within the realm of artificial neural networks (ANNs). Drawing inspiration from the structure and workings of the human brain, ANNs are computational models comprised of interconnected nodes, the so-called neurons, arranged in layers. Each neuron processes input information and transmits signals to neurons in subsequent layers.
An MLP represents a specific type of feedforward neural network, where information progresses in a singular direction, traversing from the input layer, through one or more hidden layers, and concluding at the output layer. Unlike single-layer perceptrons, which are restricted to learning linearly separable patterns, MLPs exhibit the capability to grasp intricate non-linear relationships present within input and output data.

1.2 Definition

Formally, an MLP consists of a Directed Acyclic Graph (DAG) composed of an input layer, one or more hidden layers, and an output layer. Each layer, except the input, contains one or more neurons, also called nodes or units. Neurons in adjacent layers are fully connected, meaning that each neuron in one layer is connected to every neuron in the following layer.
An MLP is operated in two phases: *forward* propagation and *backpropagation*. During the forward stage, input data is processed layer by layer, and the output of the network is computed. In backpropagation, the error between the predicted and ac-

tual outputs is calculated and used to adjust the weights of the connections between neurons according to the descending direction of the error gradient w.r.t. those, enabling the network to learn from data.
The process is repeated a variable number of times in order to obtain a model that is able fitting a desired function when given new inputs.

1.3 Applications

MLPs have found applications in various fields, including pattern recognition, classification, regression, and time series prediction.

In recent years, advancements in deep learning, a subfield of machine learning that focuses on ANNs with many hidden layers, have led to significant improvements in the performance of MLPs, becoming essential components in the creation of more complex neural network architectures.

1.4 Objective

This project aims to provide a comprehensive description of the multilayer perceptron model, including its architecture, training algorithm, and practical applications. In order to explore the low level implementation of ANNs, a small but functional library is developed using Python and employed. Through theoretical analysis and experiments on some common datasets, we seek to analyze the capabilities and limitations of MLP.

Chapter 2

Mathematical Formulas for the Multilayer Perceptron

2.1 Introduction

The multilayer perceptron (MLP) is a powerful neural network model capable of learning complex patterns in data. Understanding the mathematical foundations behind the operation of MLPs is essential for gaining insights into their behavior and designing effective learning algorithms.
In this chapter, we will explore the equations that govern the forward propagation of input data through the network, the computation of output predictions and the backpropagation of errors' gradients needed to train the network.
By analyzing the underlying mathematics, we aim to provide a complete description of how the model processes information and learns from data, in order to make the implementation of our MLP class as clear as possible.

2.2 Mathematical Notation

In this section, we introduce the mathematical notation used to describe the structure and operation of a multilayer perceptron (MLP) neural network.

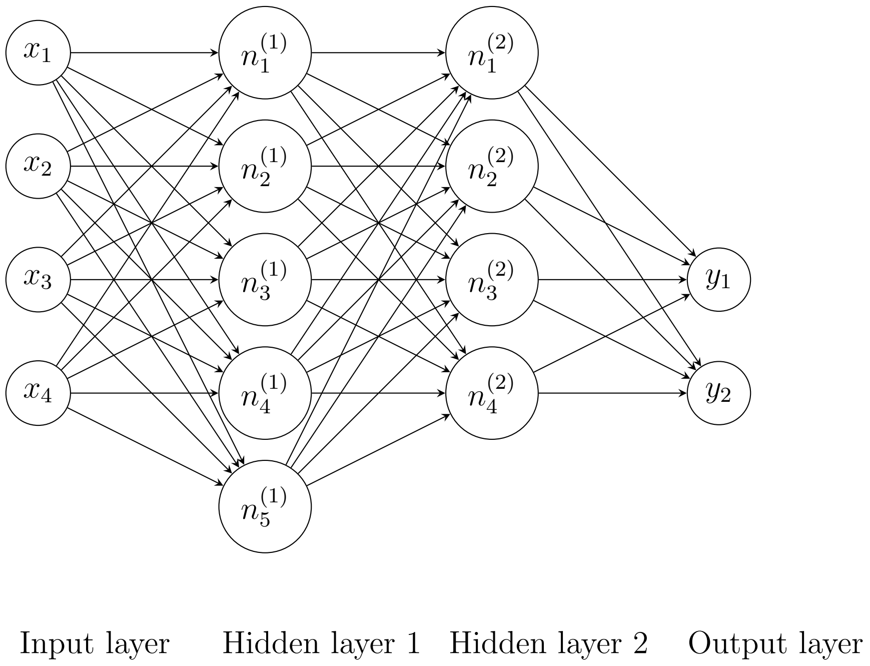

Image representing a MLP with two hidden layers

2.2.1 Layer Indexing

We denote the layers of the MLP by l, where $l = 0, 1, 2, \ldots, L$. With layer $l = 0$ we represent the input layer, $l = 1, 2, \ldots, L-1$ represent the hidden layers and layer $l = L$ stands for the output layer.

2.2.2 Neuron Indexing

Within each layer l, we index the neurons by i, where $i = 1, 2, \ldots, n^{(l)}$, with $n^{(l)}$ denoting the number of neurons in layer l.

2.2.3 Weight Matrices

The weights connecting neurons between layers $l-1$ and l are represented by the weight matrix $W^{(l)}$. Each element $w_{ij}^{(l)}$ of the weight matrix corresponds to the weight connecting neuron i in layer l to neuron j in layer $l-1$.

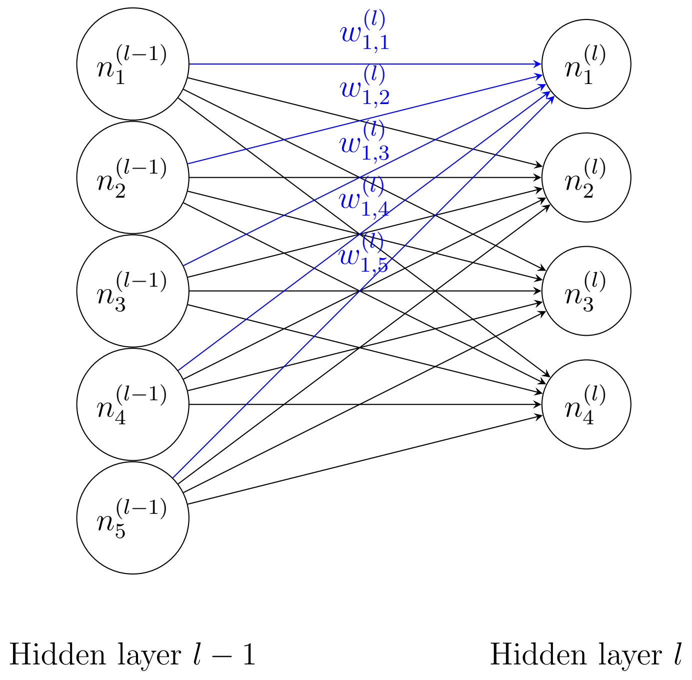

Image representing all the connections (blue) between the first neuron of layer l and all the neurons of the previous layer

2.2.4 Bias Vectors

The biases of the neurons, often kept implicit in representations, are additional connections consisting of weight-like values. In layer l they are represented by the bias vector $b^{(l)}$. Each element $b_i^{(l)}$ of the bias vector corresponds to the bias of neuron i in layer l.

2.2.5 Activation scores and Inputs

The activation scores of the neurons in layer l are denoted by the activation vector $a^{(l)}$. The inputs to the neurons in layer l are denoted by the input vector $o^{(l-1)}$ (output vector of the previous layer), with the exception of the first layer where the inputs are denoted by the vector x

2.2.6 Activation score and Output

The activation score of neuron i in layer l is denoted by $a_i^{(l)}$, and the corresponding output is denoted by $o_i^{(l)}$. These quantities are related as follows:

$$a_i^{(l)} = \sum_{j=1}^{n^{(l-1)}} W_{ij}^{(l)} o_j^{(l-1)} + b_i^{(l)}, \quad o_i^{(l)} = \sigma(a_i^{(l)})$$

where σ is the activation function applied to the activation score.

2.3 Forward Propagation

In an MLP, the forward propagation involves computing the activation scores and outputs of all the neurons in each layer and propagating the partial layer outputs forward through the network until the output layer is reached.

2.3.1 Neuron Activation

Let $o^{(l-1)}$ represent the input to layer l (output vector of previous layer), where for $l = 0$ (input layer) the input is defined with vector x. The output $o^{(l)}$ of each neuron in layer l is computed as a function of the weighted sum of the inputs and a nonlinear activation function σ:

$$o^{(l)} = \sigma(a^{(l)}), \quad a^{(l)} = W^{(l)} o^{(l-1)} + b^{(l)}$$

where $W^{(l)}$ is the weight matrix connecting layer $l - 1$ to layer l, $b^{(l)}$ is the bias vector for layer l, and σ is the activation function.

2.3.2 Output Prediction

The output prediction $\hat{y}$ of the MLP is computed as the outputs of the neurons in the layer L:

$$\hat{y} = o^{(L)}$$

where L denotes the index of the last layer.

2.4 Backpropagation

Backpropagation is the classic algorithm used to train a multilayer perceptron (and other ANNs) by iteratively adjusting the network weights to minimize the prediction error. The process involves computing the gradient of the loss function with respect to the network parameters and updating the weights using gradient descent optimization algorithm.

2.4.1 Error Calculation

The gradient of the error $\delta^{(l)}$ of each neuron in layer l is computed as the derivative of the loss function with respect to the activations of the neurons in that layer:

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial a^{(l)}}$$

where $\mathcal{L}$ is the loss function, such as the mean squared error or cross-entropy loss.

Output layer Deltas

$$\delta_i^{(L)} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial o_i^{(L)}} \frac{\partial o_i^{(L)}}{\partial a_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial o_i^{(L)}} \cdot \sigma'(a_i^{(L)})$$

Examining, for example, the case of a single input example and a loss corresponding to the mean square error, the expression can be further rewritten as:

$$\delta_i^{(L)} = \frac{\partial}{\partial o_i^{(L)}} [\frac{1}{n^{(L)}} \sum_{k=1}^{n^{(L)}} (o_k^{(L)} - y_k)^2] \cdot \sigma'(a_i^{(L)}) = \frac{2}{n^{(L)}} (o_i^{(L)} - y_i) \cdot \sigma'(a_i^{(L)})$$

Last Hidden layer Deltas

$$\delta_j^{(L-1)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L-1)}} = \sum_{i=1}^{n^{(L)}} \frac{\partial \mathcal{L}}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial a_j^{(L-1)}} = \sum_{i=1}^{n^{(L)}} \delta_i^{(L)} \frac{\partial a_i^{(L)}}{\partial a_j^{(L-1)}} = \sum_{i=1}^{n^{(L)}} \delta_i^{(L)} \frac{\partial a_i^{(L)}}{\partial o_j^{(L-1)}} \frac{\partial o_j^{(L-1)}}{\partial a_j^{(L-1)}} =$$

$$= \frac{\partial o_j^{(L-1)}}{\partial a_j^{(L-1)}} \sum_{i=1}^{n^{(L)}} \delta_i^{(L)} W_{ij}^{(L)} = \sigma'(a_j^{(L-1)}) \cdot \sum_{i=1}^{n^{(L)}} \delta_i^{(L)} W_{ij}^{(L)}$$

We can therefore derive a general expression for the delta terms of the hidden layers as:

$$\delta_j^{(l)} = \sigma'(a_j^{(l)}) \cdot \sum_{i=1}^{n^{(l+1)}} \delta_i^{(l+1)} W_{ij}^{(l+1)} \qquad \forall l \in [1, L-1]$$

2.4.2 Gradient Calculation

The network learning process consists of optimizing network weights and biases. Parameter updates are carried out by calculating the derivative of the Loss function with respect to them.

Gradient w.r.t. weights
The derivative of the loss with respect to the weight $W_{ij}^{(l)}$ can be written as:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} \cdot o_j^{(l-1)}$$

Gradient w.r.t. biases
The derivative of the loss with respect to the bias $b_i^{(l)}$ can be written as:

$$\frac{\partial \mathcal{L}}{\partial b_i^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial b_i^{(l)}} = \delta_i^{(l)} \cdot 1 = \delta_i^{(l)}$$

2.4.3 Weight Update

The weights $W^{(l)}$ and biases $b^{(l)}$ of the network are updated using gradient descent optimization algorithm. The update rule can be expressed as:

$$W^{(l)} \leftarrow W^{(l)} - \rho \frac{\partial \mathcal{L}}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \rho \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

Where ρ is the learning rate and $\frac{\partial \mathcal{L}}{\partial W^{(l)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(l)}}$ are the gradients of the loss function with respect to the weights and biases, respectively.

Chapter 3

Activation Functions and Their Derivatives

Activation functions play a crucial role in determining the output of a neuron in a neural network. In this chapter, we explore the mathematical definitions and derivatives of several commonly used activation functions, including *linear*, *sigmoid*, *ReLU*, *Leaky ReLU*, *hyperbolic tangent*, and *softmax* functions.

3.1 Sigmoid Function

The sigmoid function, also known as the logistic function, is defined as:

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

Its derivative with respect to x is given by:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

3.2 ReLU Function

The Rectified Linear Unit (ReLU) function is defined as:

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

Its derivative is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

3.3 Leaky ReLU Function

The Leaky ReLU function introduces a small slope for negative inputs to prevent the dying ReLU problem for negative inputs. It is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

where α is a small positive constant. Its derivative is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{otherwise} \end{cases}$$

3.4 Hyperbolic Tangent Function

The hyperbolic tangent function is defined as:

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

Its derivative is given by:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

3.5 Linear Function

The linear activation function (identity) simply outputs the input:

$$f(x) = x$$

Its derivative is constant:

$$f'(x) = 1$$

3.6 Softmax Function

The softmax activation function is commonly used in multiclass classification problems to convert raw output scores into probabilities. Given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the softmax function $\sigma(\mathbf{x})$ is defined as:

$$\sigma(\mathbf{x})_i = \text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad \text{for } i = 1, 2, \ldots, n$$

where e is Euler's number and x_i represents the i-th component of vector $\mathbf{x}$.

To compute the Jacobian matrix of the softmax function, we need to compute the partial derivatives of each output element with respect to each input element. Let $y_i = \sigma(\mathbf{x})_i$, then the Jacobian matrix $\mathbf{J}$ is given by:

$$J_{ij} = \frac{\partial y_i}{\partial x_j}$$

Let's compute the derivative of y_i with respect to x_j:

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right)$$

If $i \neq j$, we have:

$$\frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right) = \frac{-e^{x_i} e^{x_j}}{\left(\sum_{k=1}^{n} e^{x_k}\right)^2} = -y_i y_j$$

If $i = j$, we have:

$$\frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right) = \frac{e^{x_i}\left(\sum_{k=1}^{n} e^{x_k}\right) - e^{x_i} e^{x_j}}{\left(\sum_{k=1}^{n} e^{x_k}\right)^2} = y_i(1 - y_j)$$

Therefore, the Jacobian matrix $\mathbf{J}$ is:

$$\mathbf{J} = \begin{pmatrix} y_1(1-y_1) & -y_1 y_2 & \cdots & -y_1 y_n \\ -y_2 y_1 & y_2(1-y_2) & \cdots & -y_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ -y_n y_1 & -y_n y_2 & \cdots & y_n(1-y_n) \end{pmatrix} = y_i(\mathbf{1}\{i=j\} - y_j) \quad \forall\, i, j = 1, 2, \ldots, n$$

Where in the last equality the indicator function was used. In this context the function denoted as $\mathbf{1}\{i = j\}$, is defined as follows:

$$\mathbf{1}\{i = j\} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

Assumption:

In order to facilitate the development of the project it is assumed that the softmax activation function is used only in output layer together with categorical cross-entropy loss. This type of assumption permits us not to use the Jacobian matrix directly, while still allowing to calculate the delta terms of the backpropagation related to the loss function.

Chapter 4

Loss Functions in Gradient-Based Optimization

In gradient-based optimization, the choice of a suitable loss function is crucial for training a neural network. The loss function measures the discrepancy between the predicted output of the model and the true target values. The goal of optimization is to minimize this discrepancy by adjusting the parameters of the model using gradient descent or related optimization algorithms.

4.1 Mean Squared Error (MSE)

The mean squared error (MSE) is a common loss function used for regression tasks. It is defined as the average of the squared differences between the predicted output $\hat{y}$ and the true target values y over a dataset of N samples:

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

The MSE measures the average squared distance between the predicted and actual values, penalizing large errors more heavily.

The derivative of the MSE loss function with respect to the predicted output $\hat{y}_i$ is:

$$\frac{\partial \text{MSE}}{\partial \hat{y}_i} = \frac{2}{N}(\hat{y}_i - y_i)$$

4.2 Mean Absolute Error Loss Function

The Mean Absolute Error (MAE) is a common loss function used in regression tasks. It measures the average absolute difference between the predicted values and the actual values. Given a set of n predictions $\hat{y}_i$ and their corresponding true values y_i, the Mean Absolute Error is calculated as:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i|$$

The derivative of the MAE loss function with respect to a prediction $\hat{y}_i$ is not defined at $\hat{y}_i = y_i$, but it can be calculated using subgradients. The subgradient of the MAE loss function is defined as:

$$\frac{\partial \text{MAE}}{\partial \hat{y}_i} = \begin{cases} -1 & \text{if } \hat{y}_i < y_i \\ \text{undefined} & \text{if } \hat{y}_i = y_i \\ 1 & \text{if } \hat{y}_i > y_i \end{cases}$$

4.3 Categorical Cross-Entropy Loss

The categorical cross-entropy loss is commonly used for multi-class classification tasks. It measures the dissimilarity between the predicted class probabilities and the true one-hot encoded labels. For a dataset with N samples and K classes, the categorical cross-entropy loss is defined as:

$$\text{Categorical Cross-Entropy} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik})$$

where y_{ik} is the i-th sample's true label for class k (1 if the sample belongs to class k, 0 otherwise), and $\hat{y}_{ik}$ is the predicted probability of class k for the i-th sample.

The derivative of the categorical cross-entropy loss function with respect to the predicted output $\hat{y}_{ik}$ is:

$$\frac{\partial \text{Categorical Cross-Entropy}}{\partial \hat{y}_{ik}} = -\frac{1}{N} \frac{y_{ik}}{\hat{y}_{ik}}$$

Simplifying assumption

In order to easily calculate the backpropagation delta terms, it is assumed that this loss function is used exclusively with the softmax activation in the output layer (layer L). For a single training example, we can write the delta term of neuron j as:

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} = -\frac{\partial}{\partial a_j^{(L)}} \sum_{k=1}^{K} y_k \log(\hat{y}_k) = -\sum_{k=1}^{K} \frac{y_k}{\hat{y}_k} \frac{\partial}{\partial a_j^{(L)}} \text{softmax}(a_k^{(L)})) = -\sum_{k=1}^{K} \frac{y_k}{\hat{y}_k} \hat{y}_k(\mathbf{1}\{k=j\} - \hat{y}_j) =$$

$$= -\sum_{k=1}^{K} y_k(\mathbf{1}\{k=j\} - \hat{y}_j) = \sum_{k=1}^{K} y_k \hat{y}_j - \sum_{k=1}^{K} y_k \cdot \mathbf{1}\{k=j\} = (\sum_{k=1}^{K} y_k \hat{y}_j) - y_j = \hat{y}_j(\sum_{k=1}^{K} y_k) - y_j = \hat{y}_j - y_j$$

The generalization for the entire output layer can be written as:

$$\boldsymbol{\delta}^{(L)} = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

4.4 Binary Cross-Entropy

The binary cross-entropy loss function is commonly used in binary classification tasks. It measures the dissimilarity between the true labels and the predicted probabilities. Let y_i be the true label (either 0 or 1) and $\hat{y}_i$ be the predicted probability of class 1 (between 0 and 1) for the i-th sample.
Binary cross-entropy loss is defined as:

$$\text{Binary Cross-Entropy} = -\frac{1}{N} \sum_{i=1}^{N} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

where N is the total number of samples.

It's derivative with respect to the predicted probability $\hat{y}_i$ can be calculated as:

$$\frac{\partial \text{Binary Cross-Entropy}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i}$$

Chapter 5

Learning Rate Optimizers

5.1 Introduction

The size of the step to be taken in the descending direction of the gradient is a crucial part in reaching the convergence to the best expected minimum of the loss function. As opposed to fixing the learning rate to a given value, multiple optimization procedures can be used to adjust it dynamically on the run, resulting typically in a better and more stable behaviour.

5.2 Basic Adaptive Learning Rate

Given the entry point for the learning rate ρ, an interval is built:

$$I = [m, M] \qquad s.t. \qquad \rho \in [m, M]$$

After every update step:
if the loss function decreased, ρ is increased by a factor α, otherwise it's decreased by β, hence moving more cautiously on the descending direction.
The learning rate is kept within the interval I and typically $\alpha << \beta$

5.3 Resilient Backpropagation - RPROP

Independently from the value of ρ, the update step is determined for every parameter on the base of the sign of the gradient along that dimension.

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} + \Delta w \qquad \Delta w_t = \begin{cases} \Delta w_{t-1} - \beta & \text{if } \frac{\delta \mathcal{L}}{w_{i,j}^{(l)}} \text{ changed sign} \\ \Delta w_{t-1} + \alpha & \text{if } \frac{\delta \mathcal{L}}{w_{i,j}^{(l)}} \text{ kept sign} \end{cases}$$

Similarly to basic adaptive learning rate, the step is taken more cautiously when the loss crosses a minimum in the dimension associated to a parameter, hence typically $\alpha << \beta$.

5.4 Adaptive Moment Estimation - ADAM

The Adam optimizer is a popular optimization algorithm commonly used in training deep neural networks. It stands for "Adaptive Moment Estimation" and combines ideas from two other popular optimization algorithms: RMSprop and Momentum. It computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

Given a parameter θ, its gradient g_t at time step t, and hyperparameters β_1, β_2, and ϵ, the Adam update rule for the parameter θ is as follows:

$$\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2} \\ \theta_{t+1} &= \theta_t - \rho \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned}$$

where:

- m_t and v_t are estimates of the first and second moments of the gradients respectively.
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates.
- ρ is the learning rate.

- β_1 and β_2 are exponential decay rates for the moment estimates.
- ϵ is a small constant added to the denominator for numerical stability.

In the context of our project, given a weight (or bias) parameter, the update rule is defined as follows:

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} + \Delta w \qquad \Delta w = -\rho \tfrac{m_{i,j}}{\sqrt{v_{i,j}}+\epsilon}$$

Chapter 6

Code Analysis

6.1 Data structures

Handling the tasks needed for an efficient training of an ANN requires the possibility of saving data structures such as Vectors and Matrices or more in general *Tensors*. While still perfectly possible, element wise operations can frequently be substituted with tensorial operations. With the help of parallelism on modern hardware this results in faster execution times and more readable coded algorithms.

6.2 Libraries

In order to deal efficiently with the aforementioned data structures and the needed operations, the *NumPy* library is employed in a Python environment. This enables the instantiation of flexible tensors and, exploiting multi-threaded approaches, grants swift performances for typically expensive routines.
Apart from basic Python functionalities, few other modules are used for ease of saving trained models (*Pickle*), plotting results (*Matplotlib*) and loading complex datasets (*Deeplake*).

6.3 MLP class

One all-inclusive class for a Multilayer Perceptron is built in a single file. Taking advantage of NumPy data structures and a flexible constructor, the network architecture is built using layer-specific activation functions and the needed *Gradients*, *Activation scores* and *Outputs* containers are instantiated to be reused in various

methods.
The MLP object includes functions to start a given number of *epochs* of training with the selected loss measure and optimizer (Basic Adaptive lr, RPROP, ADAM) on a given dataset, indicating with X and Y the inputs and expected results (targets) respectively. If provided, an inference is also produced on the *Validation set*, and the results can be plotted both online and offline. The default training procedure takes into account the whole training set as a single batch, this can be further subdivided in parts. The training can finally be early stopped considering the validation set loss given a number of *"patient"* epochs to wait.
Other methods include the possibility of saving and loading the trained models if compatible with the current architecture and calculating the accuracy, precision, recall and F1 scores along with the *Confusion Matrix* associated with a classification problem.

6.4 Algorithms

A Multi Layer Perceptron, as every ANN, needs the two basic forward and backward procedures to enhance it's parameters' values and fit the provided data in case of supervised learning. Calculating the activations and outputs in the forward stage, the MLP accounts for the respective gradient's components of weights and biases during the backward, applying the Backpropagation algorithm.
The forward propagation method is reused every time the model needs to give a prediction either during the training or when making inference on test data.
In the process of adjusting the network parameters and depending on the user choice, various optimization techniques take place to model the size of the gradient (Gradient clipping to prevent explosion) and the learning rate.

Chapter 7

Experiments

7.1 Environment

To test the flexibility of the MLP class, multiple different files are used to employ our Network on *Classification* and *Regression* problems, probing the performances of optimizations and functionalities.

Starting from the classical, non-linearly separable, *XOR* problem, the class is then used to fit data from two commonly used datasets:

- Boston Housing - Scalar Regression
- Wine - Multiclass Classification

Furthermore, with the goal of testing efficiency on computationally expensive trainings, *MNIST* dataset is employed, reshaping it's inputs into vectorial representations.

7.2 Boston Housing

The dataset poses the problem of estimating the value of houses given respective features (13 in total) such as *per capita crime rate*, *average number of rooms per dwelling* and *nitric oxides concentration.*
The employed ANN is expected to be able to output a scalar value for every set of the 13 indicators.
For this purpose the inputs are first normalized, centering each feature w.r.t. the

overall mean and dividing by the standard deviation. The architecture includes hidden layers with non-linear activation functions, paired with a single linear output unit.

7.3 Wine

Consisting of only 178 instances, this dataset poses the problem of classifying 3 different varieties of wine, based on the 13 chemical constituents given.
With it's custom architecture, the network applies a softmax activation to the 3 output neurons, inferring the probability distribution of the classes, to be compared with the one-hot encoded targets.

7.4 MNIST

Used as benchmark in the recognition of hand-written digits, MNIST dataset contains two portions (Training & Validation) of 28x28 tensorial represented images on a single grayscale level (0, 255).
Commonly, image classification tasks are better suited for Convolutional Neural Networks, but at this resolution, sufficient results can still be obtained flattening the input matrices into vectors, then fed to an MLP. To further improve the learning, inputs are normalized to values between 0 and 1 to stabilize the module of the activation scores. A *Gridsearch* algorithm is then performed on multiple Network hyperparameters to narrow the scope of suitable solutions.
Results and details of the best performing architectures are reported below.

MNIST best model:
Architecture: $784 \to 900 \to 1000 \to 200 \to 10$
Activation functions: *sigmoid - sigmoid - sigmoid - softmax*
Starting learning rate: 0.0001
Optimizer: *ADAM*
Batch size: 3000

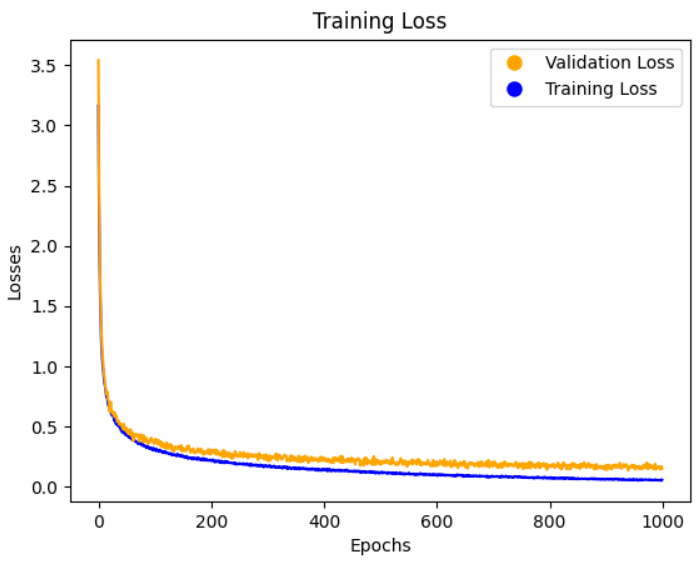

Figure 7.1: Best training plot - MNIST

Class 1:

Accuracy = 0.9868
Precision = 0.9231536926147704
Recall = 0.9438775510204082
F1 = 0.9334006054490414

Class 2:

Accuracy = 0.9927
Precision = 0.9609375
Recall = 0.9753303964757709
F1 = 0.9680804547442063

Class 3:

Accuracy = 0.9723
Precision = 0.8719211822660099
Recall = 0.8575581395348837
F1 = 0.8646800195407914

Class 4:

Accuracy = 0.971
Precision = 0.8495145631067961
Recall = 0.8663366336633663
F1 = 0.8578431372549019

Class 5:

Accuracy = 0.9724
Precision = 0.8508946322067594
Recall = 0.8716904276985743
F1 = 0.8611670020120724

Class 6:

Accuracy = 0.9695
Precision = 0.8354285714285714
Recall = 0.8195067264573991
F1 = 0.8273910582908884

Class 7:

Accuracy = 0.9828
Precision = 0.9145569620253164
Recall = 0.9050104384133612
F1 = 0.9097586568730325

Class 8:

Accuracy = 0.9789
Precision = 0.9181166837256909
Recall = 0.872568093385214
F1 = 0.8947630922693266

Class 9:

Accuracy = 0.9665
Precision = 0.827021494370522
Recall = 0.8295687885010267
F1 = 0.8282931829830855

Class 10:

Accuracy = 0.9677
Precision = 0.8369351669941061
Recall = 0.844400396432111
F1 = 0.8406512086827824

Chapter 8

Conclusions

The scope of the project, as initially intended, led to the development of a light-weight Multilayer Perceptron Class, flexible enough to fit the experiment's requirements and evaluating the learning capabilities of a given Network.
Our class includes a good variety of the key-features typically employed by an MLP, ranging from learning rate optimizers, like RPROP and ADAM, to early stopping criterion and evaluation metrics computation (Accuracy, Precision, Recall, F1) along with confusion matrices for classification problems.
Given the modular structure of the code, future work may focus on the extension of the functionalities as well as the use of the architectures as building blocks in more complex models.

In conclusion, the project enabled us to delve into the low level math details behind Neural Networks' learning problems, granting the ability to scale up our knowledge to more advanced Machine learning topics.