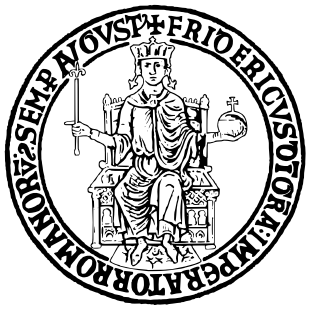
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Corso di Neural Network and Deep Learning

Building a Neural Network from scratch

Prof
Roberto Prevete

Candidato
Giovanni Falcone N97/0451

Anno Accademico 2023-2024

Contents

Chapter 1

Project description

Part A

Progettazione ed implementazione di una libreria di funzioni per:

- simulare la propagazione in avanti di una rete neurale multi-strato full-connected. Con tale libreria deve essere possibile implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascun strato
- la realizzazione della back-propagation per reti neurali multi-strato, per qualunque scelta della funzione di attivazione dei nodi della rete e la possibilità di usare almeno la somma dei quadrati o la cross-entropy con e senza soft-max come funzione di errore.

Part B

Si consideri come input le immagine raw del dataset mnist. Si ha, allora, un problema di classificazione a C classi, con C=10. Si estragga opportunamente un dataset globale di N coppie, e lo si divida opportunamente in training e test set (Considerare almeno 10000 elementi per il training set e 2500 per il test set). Si fissi la discesa del gradiente con momento come algoritmo di aggiornamento dei pesi, si studi l'apprendimento di una rete neurale (ad esempio epoche necessarie per l'apprendimento, andamento dell'errore su training e validation set, accuratezza sul test) con un solo strato interno di neuroni al variare di eta (learning rate) e del momento per almeno 5 diverse dimensioni (numero di nodi) dello strato interno. Scegliere e mantenere invariati tutti gli altri "parametri" come, ad esempio, le funzioni di output.

Chapter 2

Part A

This chapter covers the implementation aspects of the library, specifically focusing on its structure and the implementation of the main algorithms.

2.1 Library strcture

The library structure is as the follows:

/
- dataset
  - dataset.py
  - mnist_test.csv
  - mnist_train.csv
- evaluation
  - $learning\_mode_i$
    - $model_i$
    - tuning.xlsx
- plot
  - $learning\_mode_i$
    - test
- utility
  - utility.py
- network
  - early_stopping.py
  - loss_functions.py
  - activation_functions.py
  - grid_search.py
  - neural_network.py
- model_selection.py
- test.py
- neural_network_from_scratch.ipynb

A brief description for each file/folder is as follows:

- **dataset** contains the training set and the test set in CSV format as well as

a python file that contains function to retrieve both sets, obtain one-hot encoded labels, etc;

- **evaluation** contains a directory $learning\_mode_i$ that represents the learning modality chosen to train the network. This directory contains a folder for each combination of hyperparameters to evaluate. Each directory contains the `model.pkl`, the plot of accuracy and loss for both training and validation set and a report (f1-score, accuracy, etc).
  The `tuning.xlsx` is an excel file that contains the informations of the various trained models (learning rate, momentum, number of neurons, accuracy, number of epochs, etc.);
- **plot** is the directory which contains the evaluation of best models. It is further divided in three different directory (one for each type of learning). Each directory, contains a folder which contains the results of testing. Specifically, it includes a classification report and a confusion matrix;
- **utility** contains utility functions (e.g get a random element from a set and plot the image associated to it);
- **network** is the main folder which contains the neural network library;
  - **activation_functions.py**, as the name suggests, contains all activation funtions. Specifically, it is a class that includes static methods such as `tanh`, `sigmoid`,`soft_max`, etc;
  - **loss_functions.py** is a class that contains static mathods such as `cross_entropy`, etc;
  - **early_stopping.py** is a class that defines all early stopping criteria (UP, Generalization Loss, Progress Quotient, patience);
  - **grid_search.py** is a class that trains as many models as there are combinations of hyperparameters to figure out which one is best;
  - **neural_network.py** is the class that defines the network. Therefore, it includes the weights and biases initialization for each layer, how many hidden layers, the train function, accuracy function, forward and back-propagation functions, etc;
- **model_selection.py** a script used to apply the grid search;
- **test.py** is a script which can be used to test the best model provided in evaluation phase;
- Finally, there is a jupyter notebook file in which a use case is shown.

2.2 Dataset

Load the dataset

```
@statitmethod
def load_mnist()
```

The purpose of this function is to load the MNIST dataset. After normalizing both training and test sets, it returns the images and their corresponding labels, with the labels in one-hot encoded format.

Returns

`train_imgs`: `np.ndarray`
The matrix of digits that should be used for training. The matrix has a shape of (784, 60000) where 784 is the number of pixel (features) and 60000 is the number of samples.

`train_labels`: `np.ndarray`
The one-hot encoded matrix, which has a shape of (10, 60000) where 10 is the number of classes and 60000 is the number of samples.

`test_imgs`: `np.ndarray`
The matrix of digit images that should be used for testing the network. The matrix has a shape of (784, 10000).

`test_labels`: `np.ndarray`
The one-hot encoded matrix of labels used for testing. It has a shape of(10, 10000).

Splitting training set in training and validation sets

```
@staticmethod
def train_val_split(data, target, percentage=0.2, random_state=None, ↩
    shuffle=True)
```

This function is used in order to split the training set into training and validations set, based on a certain percentage.

Parameters

`data`: `np.ndarray`
The training set organized as matrix (d, N) where d is number of features and N is the number of samples.

`target`: `np.ndarray`
The one-hot encoded matrix used for training.

`percentage`: `float`, *Default=0.2*
The percentage used to split the training set(e.g 80% for training and 20% for validation).

`random_state`: `int`, *Default=None*

	Controls the shuffling applied to the data before applying the split. Pass an int for reproducible output across multiple function calls.
`shuffle`:	`bool`, *Default=True* Whether or not to shuffle the data before splitting.

Returns

`x_train`:	`np.ndarray` The new training set of images with a shape of $(d, N - percentage)$, where d is the number of features.
`y_train`	`np.ndarray` The new one-hot encoded matrix of labels with a shape of $(c, N - percentage)$, where c is the number of target classes.
`x_val`:	`np.ndarray` The validation set of images with a shape $(d, percentage)$.
`y_val`:	`np.ndarray` The one-hot encoded matrix of labels used for validation with a shape $(c, percentage)$.

Errors

Raise a (`ValueError`) if `data` or `target` aren't numpy arrays or if their shape is not 2. (d, N).

2.3 Utility class

Print dataset

```
@staticmethod
def print_info_dataset(x_tr_shape, y_tr_shape, x_val_shape,
                       y_val_shape, x_te_shape, y_te_shape)
```

Prints shapes of training set, validation set and test set in a tabular format.

Print network

```
@staticmethod
def print_info_network(
        input_size,
        n_hidden_layer,
        hidden_neurons,
        weights_shape,
        output_size,
        activation_function,
        error_function
```

```
        ) -> None:
```

Prints in tabular format the informations about the network such as number of neurons for input, hidden and output layer, the activation function of hidden and output layers, the error function chosen and the shape of the matrix of the weights for each layer.

Get a random element

```
    @staticmethod
    def get_random_elem(test, target, plot=True)
```

The function take a random element from test-set and returns it with its label.

Parameters

`test`: `np.ndarray`
A matrix with a shape of (d, N) where d is the number of *features* and N is the number of samples.

`target`: `np.ndarray`
The one-hot encoded matrix with a shape of (c, N) where c is the number of *target classes* and N is the number of samples.

`plot`: `bool`, *Default=True*
If $True$, it shows the image of chosen digit.

Returns

`elem`: `np.ndarray`
The element chosen. It's a matrix with a shape of $(d, 1)$ where d is the number of *features*.

`gold`: `int`
The label of the chosen element.

Errors

Raise a `ValueError` if `test` or `target` aren't numpy arrays or if their shape is not 2.

Split_batch

```
    @staticmethod
    def split_batch(Y, k)
```

It breaks Y (one-hot encoded matrix of labels) into k subsets, representing mini-batches, and distributes the elements equally to each batch.

Parameters

`Y`: `np.ndarray`
The one-hot encoded matrix of labels with a shape of (c, N) where c is the number of *target classes* and N is the number of samples.

`k`: `int`
Number of mini-batches

Returns

`indY`: `List[np.ndarray]`
A list of k numpy arrays. Each array contains the indexes of the elements that belonging to a certain class.

Errors

Raise a `ValueError` if `Y` isn't numpy array with a shape of (x, y) or the value of k isn't greater than 1.

2.4 Activation class

This class defines all possible activation functions that each layer of a neural network can use. All methods are marked as `static` since they don't relay on any instance. Instead, they operate directly on the input parameters provided, making them suitable for a static context

Identity

```
@staticmethod
def identity(x, der = 0):
```

Compute the activation value of m neurons using identity function. Return identity function if $der = 0$, derivative of identity function otherwise (i.e 1).

Parameters

`x`: `np.ndarray`
A matrix with a shape of (m, N) where m is the number of *neurons* and N is the number of samples.

`der`: `int`, *default=0*
A flag to indicate whether to return the activation function or the derivative (for learning phase). If $der = 0$ returns the identity, the derivatives otherwise.

Returns

`x`: `np.ndarray` (`int` if *der* > 0)
the same matrix if $der = 0$, 1 otherwise.

Sigmoid

```
@staticmethod
def sigmoid(x, der = 0):
```

Compute the activation value of m neurons using sigmoid function: it can take any real-valued number and map it into a value between 0 and 1. Return sigmoid function if $der = 0$, derivative otherwise.

Parameters

`x:` `np.ndarray`
A matrix with a shape of (m, N) where m is the number of *neurons* and N is the number of samples.

`der:` `int`, *default=0*
A flag to indicate whether to return the activation function or the derivative (for learning phase). If $der = 0$ returns the sigmoid, the derivatives otherwise.

Returns

`x:` `np.ndarray`
The result of sigmoid function (resp. derivative).

tanh

```
@staticmethod
def tanh(x, der = 0):
```

Compute hyperbolic tangent element-wise. Return tanh function if $der = 0$, derivative otherwise $(1 - tanh(x)^2)$.

Parameters

`x:` `np.ndarray`
A matrix with a shape of (m, N) where m is the number of *neurons* and N is the number of samples.

`der:` `int`, *default=0*
A flag to indicate whether to return the activation function or the derivative (for learning phase). If $der = 0$ returns the tanh, the derivatives otherwise.

Returns

`x:` `np.ndarray`
The corresponding hyperbolic tangent values (resp. derivative).

ReLU

```
@staticmethod
def relu(x, der = 0):
```

Compute the activation value of m neurons using ReLU (Rectified Linear Unit) function element wise. It's defined as $max(0, x)$. Returns ReLU function if $der = 0$, derivative otherwise.

Parameters

x: `np.ndarray`
A matrix with a shape of (m, N) where m is the number of *neurons* and N is the number of samples.

der `int`, *Default=0*
A flag to indicate whether to return the activation function or the derivative (for learning phase). If $der = 0$ returns the ReLU, the derivatives otherwise.

Returns

x: `np.ndarray`
The result of ReLU function (resp. derivative).

Leaky ReLU

```
@staticmethod
def l_relu(x, der = 0):
```

Compute the activation value of m neurons using leaky ReLU function element-wise. Return leaky relu function if $der = 0$, derivative otherwise. Formula:
$f(x) = alpha * x$ if $x \leq 0$
$f(x) = x$ if $x > 0$

Parameters

x: `np.ndarray`
A matrix with a shape of (m, N) where m is the number of *neurons* and N is the number of samples.

der: `int`, *Default=0*
A flag to indicate whether to return the activation function or the derivative (for learning phase). If $der = 0$ returns the Leaky ReLU, the derivatives otherwise.

Returns

x: `np.ndarray`
The result of leaky relu function (resp. derivative).

Softmax

```
@staticmethod
def softmax(x, der = 0):
```

Compute the activation value of m neurons using softamx function. Returns a probability distribution. The *softmax function* is defined as follows:

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_k e^{x_k}}$$

Parameters

y `np.ndarray`
A matrix with a shape of (c, N) where c is the number of *target classes* (10 in our case) and N is the number of samples.

Returns

`z` `np.ndarray`
A probability distribution organized as a matrix with a shape of (c, N).

2.5 Loss class

This is a class that defines all the loss functions used to quantify the level of error in the model's prediction. Therefore it defines the **Cross entropy** and **Sum of squares** functions.

All methods are marked as `static` since they don't relay on any instance. Instead, they operate directly on the input parameters provided, making them suitable for a static context.

Softmax Cross-entropy

```
@staticmethod
def cross_entropy_softmax(y_pred, y_true, der=0)
```

It's used for classification problems.
It computes the cross-entropy applying softmax to the prediction.

Parameters

`y_pred:` `np.ndarray`
The prediction of the network (more specifically, raw scores before applying softmax). Thus, is a matrix with a shape of (c, N) where c is the number of target classes and N is the number of samples.

`y_true:` `np.ndarray`
A matrix with a shape of (c, N) of one-hot encoded true class labels.

`der:` `int`, *Default=0*
A flag to indicate whether to return the loss or the derivative. If $der = 0$ returns the loss, the derivatives of the error function with respect the output otherwise.

Returns

`cross entropy:` `float`
If $der = 0$, a value that quantifies the level of error in the model's prediction. If $der \neq 0$ the derivative of the error function with respect to the predictions.

Cross-entropy

```
@staticmethod
def cross_entropy(y_pred, y_true, der = 0, epsilon=1e-15)
```

Compute cross-entropy given predictions and one-hot encoded ground truth labels.

Parameters

`y_pred`: `np.ndarray`
It's a matrix with a shape of (c, N) where c is the number of target classes and N is the number of samples. The values represent the predicted class for each sample.

`y_true`: `np.ndarray`
It's a matrix with a shape of (c, N) of one-hot encoded true class labels.

`der`: `int`, $Default = 0$
A flag to indicate whether to return the loss or the derivative. If $der = 0$ returns the loss, the derivatives of the error function with respect the output otherwise.

`epsilon` `float`, $Default = 1e - 15$
It's a constant to clip prediction to avoid taking log of zero, which would result in undefined values.

Returns

`cross entropy` `float`
If $der = 0$, a value that quantifies the level of error in the model's prediction. If $der \neq 0$ the derivative of the error function with respect to the predictions.

Sum of squares

```
@staticmethod
def sum_of_squares(y_pred, y_true, der = 0)
```

Loss function for regression problems; it returns the sum of squares if $der = 0$, derivative of loss function with respect to the predictions otherwise.

Parameters

`y_pred`: `np.ndarray`
It's the prediction of the network. Thus, is a matrix with a shape of (c, N) where c is the number of target classes and N is the number of samples.

`y_true` `np.ndarray`
A matrix with a shape of (c, N) containing the true target values.

`der`: `int`, $Default = 0$
A flag to indicate whether to return the loss or the derivative. If $der = 0$ returns the loss, the derivatives of the error function with respect the output otherwise.

Returns

`sum of squares` `float`
If $der = 0$, a value that quantifies the level of error in the model's prediction. If $der \neq 0$ the derivative of the SSE with respect to the predictions.

2.6 GridSearch class

This class trains as many models as there are input combinations to find the best hyperparameters.

Constructor

```
def __init__(self, model_class, param_grid)
```

Instantiates the `GridSearch` object.

Parameters

`model_class:` `function`
The model class. In this case can be only `NeuralNetwork`.

`param_grid` `dict`
The combination of hyperparameters. An example of dictionary is:

```
param_grid = {
    "n_hidden_layers": [1],
    "activation_list": [[↩
ActivationFunctions.sigmoid, ↩
ActivationFunctions.identity]],
    "error_function": [Loss.↩
cross_entropy_softmax],
    "m_neurons_list": [20, 50, 100,↩
 200, 500],
    "learning_rate": [0.0002, ↩
0.0001, 0.00009, 0.00005],
    "momentum": [0.5, 0.75, 0.9]
}
```

Train

```
def train(self,
        x_train: np.ndarray,
        y_train: np.ndarray,
        x_val: np.ndarray,
        y_val: np.ndarray,
        epochs: int,
```

```
            mode='batch',
            num_mini_batches = 32,
            early_stopper = EarlyStopping(),
            f1_avg_type = None)
```

This function creates at each iteration (given the number of combinations to be tested) a `NeuralNetwork` object and invokes the `train` method of its class (`NeuralNetwork`) by passing it the parameters of the current combination.

For each combination it create a directory with the ID of corresponding combination. Each directory contains a plot of loss and accuracy, the `model.pkl` and a report (f1-score, accuracy, etc).

Once all combinations have been tried, it creates and saves an excel file with all training data with the corresponding combination.

Parameters

The same parameters of method `train` of the class `NeuralNetwork` (see 5).

Returns

The same parameters of method `train` of the class `NeuralNetwork` (see 5).

How to get the best model

```
def best_param(self, metric='acc_val'):
```

Returns the model with higher specified metric.

Parameters

`matric`: `string`, *Default='acc_val'*
The metric chosen to evaluate the model. It can assume one of this values: [*'acc_val'*, *'acc_train'*, *'f1_train'*, *'f1_val'*]

Returns

`model`: `dict`
The information of best model according to the specified metric.

Errors

Raise a `ValueError` if `metric` has not the right value.

2.7 Early stopping class

This class allows the client to choose from several early stopping criteria. These criteria are based on Prechelt paper [1]. Therefore, the early stopping criteria are:

- Patience
- Generalizazion Loss

[1] Prechelt, Lutz. "Early stopping-but when?." Neural Networks: Tricks of the trade. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. 55-69.

- Progress Quotient
- UP

Constructor

```
def __init__(self, patience=15, strip=0, alpha=0.1,
             type_early_stopping='patience')
```

Instantiates the `EarlyStopping` object.

Parameters

`patience`:	`int`, *Default=15* A value representing after how many epochs to stop training if the validation error does not decrease.
`strip`	`int`, *Default=0* The value of strip (the number of errors to consider) used for UP and PQ.
`alpha`:	`float`, *Default=0.1* The value used for GL and PQ. If their value are greater than alpha it stops learning.
`type_early_stopping`:	`String`, *Default='patience'* The criteria to use. It can assume one of this values: $'patience', 'GL', 'PQ', 'UP'$

Errors

Raise a `ValueError` if one of this conditions is not satisfied:

- `patience` must be an integer ≥ 0
- `type_early_stopping` must assume one of this values: $'patience', 'GL', 'PQ', 'UP'$
- when the criteria is $'GL', 'PQ'$ or $'UP'$ `strip` must be an integer greater than 0.

check_early_stop_condition

```
def check_early_stop_condition(self, epoch, e_train, e_val_curr, ↩
    e_val_min, flag):
```

This main function of the `EarlyStopping` class. It handles the early stop condition. Therefore, it computes PQ, UP, GL or it handle the $patience$ counter based on $type\_early\_stopping$.

Parameters

epoch: `int`
The current epoch.

e_train `float`
The current training error.

e_val_curr: `float`
The current validation error.

e_val_min: `float`
The smallest validation error up to that time.

flag: `bool`
Used to figure out when to decrease the patience counter. Decrease the counter if $flag = True$.

Returns

res `bool`
If $True$ the early stopping condition is verified (i.e stop learning), $False$ otherwise.

handle_patience

```
def __handle_patience(self, flag)
```

Decrease the patience counter if `flag` is $True$. Returns $True$ when the counter has reached 0.

Parameters

flag: `bool`
Used to figure out when to decrease the patience counter. Decrease the counter if $flag = True$.

handle_UP

```
def __handle_UP(self, e_val_curr):
```

This function is used when $type\_early\_stopping = 'UP'$. It stops when the generalization error increased in s successive strips. It applies the following rule:

UP_s : stop after epoch t iff UP_{s-1} stops after epoch $t-k$ and $E_{va}(t) > E_{va}(t-k)$
UP_1 : stop after first end-of-strip epoch t with $E_{va}(t) > E_{va}(t-k)$

Parameters

e_val_curr: `float`
The current validation error.

Returns

res `bool`
If $True$ the early stopping condition is verified (i.e stop learning), $False$ otherwise.

handle_GL

```
def __handle_generalization_loss(self, e_val_curr, e_val_min)
```

The function computes $GL = 100 * (\frac{E_{val}(t)}{E_{opt}} - 1)$, where t is the current epoch and E_{opt} is the minimum validation error up to that time. Returns True if $GL > \alpha$ (defined in the constructor) and if the condition of early stopping is 'GL'.

Parameters

`e_val_curr`: `float`
The current validation error.

`e_val_min`: `float`
The smallest validation error up to that time.

Returns

`res` `bool`
If $True$ the early stopping condition is verified (i.e $GL > \alpha$), $False$ otherwise.

handle_PQ

```
def __handle_progress_quotient(self, e_train)
```

It represents how much was the average training error during the strip larger than the minimum training error during the strip. It computes:

$$P_k(e) := 1000 \cdot \left(\frac{\sum_{e'=e-k+1}^{e} E_{tr}(e')}{k} \cdot \frac{1}{\min_{e'=e-k+1}^{e} E_{tr}(e')} - 1 \right)$$

$$PQ = \text{ stop when } \frac{GL(e)}{P_k(e)} > \alpha$$

Parameters

`e_train`: `float`
The current train error.

Returns

`res` `bool`
If $True$ the early stopping condition is verified (i.e $\frac{GL(t)}{P_k(t)} > \alpha$), $False$ otherwise.

2.8 Neural Network class

This is the main class of the library that allows to create a neural network with one or more internal layers that can be trained (and validated) and finally tested.

2.8.1 Constructor

The constructor of `NeuralNetwork` is defined by the following attributes:

- **input_size** (`np.ndarray`): it represents the input layer of the network, which means how many neurons have the input layer (e.g for MNIST input is equal to 784). Therefore, it represents the number of features.
- **ouput_size** (`np.ndarray`) : it represents the ouput layer (i.e the number of target classes).
- **n_layers** (`int`): is an integer that represents the number of hidden layer plus the output layer.
- **m_neurons** (`int`): is a list of integers where each element represents the number of neurons in the corresponding layer. The size of this list must be equal to `n_layers - 1`, otherwise a `ValueError` will be raised.
- **activation_list** (`List[ActivationFunction]`): is a list where each element is an `ActivationFunction` object corresponding to a layer. The size of this list must be equal to `n_layers`, otherwise a `ValueError` will be raised.
- **error_function** (`LossFunction`): it represents the `LossFunction` object used to compute the loss function.
- **weights** (`np.ndarray`): is a list where each element is a matrix of a shape (m, n) where m is the number of neurons in that layer and n is the number of neurons of previous layer. `n_layers` matrices will be generated randomly using a normal distribution with a mean of 0 and a standard deviation of 0.1.
- **biases** (`np.ndarray`): similar to `weights` but the matrix has a shape of $(m, 1)$ where m is the number of neurons of the current layer.
- **velocity_w** and **velocity_b** are the accumulated velocity used for gradient descent with momentum as updated rule.

For example, a `NeuralNetwork` with 2 hidden layers is instantiated in the following way:

```
class NeuralNetwork:
    def __init__(self, input_size, output_size, n_hidden_layers, ↩
    m_neurons_list, activation_list, error_function):
        # check if ValueError
        # ...

        self.input_size = input_size
        self.output_size = output_size
        self.n_layers = n_hidden_layers + 1 # + 1 -> output layer
        self.m_hidden_neurons_list = m_neurons_list
        self.activation_list = activation_list
```

```
            self.error_function = error_function
            self.weights = []
            self.biases = []
            self.velocity_w = [0] * self.n_layers
            self.velocity_b = [0] * self.n_layers

            # init weights and biases
            self.__init_parameters()

    # initialize the network
    nn = NeuralNetwork(input_size=X_train.shape[0],
                       output_size=Y_train.shape[0],
                       n_hidden_layers=2,
                       m_neurons_list=[50, 35],
                       activation_list=[ActivationFunctions.sigmoid,
                       ActivationFunctions.sigmoid,
                       ActivationFunctions.identity],
                       error_function=Loss.cross_entropy_softmax)
```

2.8.2 Other methods

Summary

```
def summary(self):
```

Prints all network information in tabular format. It prints the size of input and output layers, number of hidden layer, the shape of the matrix of the weights and biases for each layer, the activation function for each layer and the error function used.

Copy network

```
def copy_network(self):
```

Return a copy of network using `deepcopy` function.

Compute accuracy

```
def __compute_accuracy(self, y_net, target)
```

This function computes the accuracy of the neural network's predictions. First, it applies the `softmax` function to the network's output (result of `forward_propagation`) to convert raw scores into probabilities. Then, for each example, it compares the predicted class (the index of the maximum probability) with the "gold" value (the true class label).

Finally, it sums all *True* values and then it divides the result by the number of samples.

Parameters

`y_net:` `np.ndarray`
A matrix with a shape of (c, N) where c is the number of *target classes* (10 in our case) and N is the number of samples. It's the output of the network.

`target:` `np.ndarray`
The one-hot encoded matrix which contains the true labels.

Returns

`accuracy:` `float`
A float value between 0 and 1.

Evaluate model

```
def evaluate_model(self, test, target)
```

This function makes a prediction for each sample of test set. It computes the accuracy, the confusion matrix and a report containing $F1-score$, $Recall$ and $Precision$.

Parameters

`test:` `np.ndarray`
It's the test set. It must have a shape of (s, N) where s is the number of features and N is the number of samples.

`target:` `np.ndarray`
Ground truth (correct) labels. Thus, is the one-hot encoded matrix which contains the true labels. It must have the same shape of `test` parameter.

`target:` `List[str]`
List of labels to index the confusion matrix.

Returns

`accuracy:` `float`
Return the fraction of correctly classified samples (float).

`cm:` `np.ndarray`
Confusion matrix with a shape of $(n\_classes, n\_classes)$, whose $i-th$ row and $j-th$ column entry indicates the number of samples with true label being $i-th$ class and predicted label being $j-th$ class.

`report:` `dict`
A dictionary which represents a text summary of the precision, recall, F1 score for each class. Dictionary has the following structure:

```
{'label 1': {'precision':0.5,
             'recall':1.0,
             'f1-score':0.67,
             'support':1},
 'label 2': { ... },
  ...
}
```

Errors

Raise a `ValueError` if `test` or `target` aren't numpy arrays or if their shape is not 2.

Predict

```
def predict(self, input)
```

This function makes a single prediction given a certain input. Thus, it applies the forward propagation using the input and converts the row score in probabilities using the softmax function.

Parameters

`input:` `np.ndarray`
A single example. It must be a matrix of shape $(d, 1)$ where d is the number of features.

Returns

`prediction:` `np.ndarray`
A matrix of shape $(c, 1)$ where 'c' is the number of classes. Each row represents the probability of belonging to a certain class.

Errors

Raise a (`ValueError`) if `input` does have not a shape of $(d, 1)$ or if it's not a numpy array.

get_train_val_accuracy

```
def __get_train_val_accuracy(self, y_pred_train, y_train,
                             y_pred_val, y_val, train_ac_list,
                             val_ac_list)
```

It computes the accuracy of training set and validation set. Specifically, given the network prediction of both training and validation set, it calculates the accuracy and inserts it into the respective list.

Parameters

`y_pred_train`:	`np.ndarray` The network prediction on training set (i.e the forward propagation's result applied on the training set). It has a shape of (c, N) where c is the number of target classes and N is the number of samples.
`y_train`:	`np.ndarray` The one-hot encoded matrix which contains the true labels. It represents the labels of training set. It has the same shape of `y_pred_train`.
`y_pred_val`:	`np.ndarray`) The network prediction on validation set (i.e the forward propagation's result applied on the validation set). It has a shape of (c, N) where c is the number of target classes and N is the number of samples.
`y_val`:	`np.ndarray` The one-hot encoded matrix which contains the true labels. It represents the labels of validation set. It has the same shape of `y_pred_val`.
`train_ac_list`:	`List[float]` A list where each element represents the accuracy on the training set of a certain epoch.
`val_ac_list`:	`List[float]` A list where each element represents the accuracy on the validation set of a certain epoch.

Returns

`train_ac_list`:	`List[float]` The updated list where the last element is the accuracy on training set of the current epoch.
`val_ac_list`:	`List[float]` The updated list where the last element is the accuracy on validation set of the current epoch.

get_train_val_loss

```
def __get_train_val_loss(self, y_pred_train, y_train, y_pred_val,
                         y_val, train_loss_list, val_loss_list)
```

It computes the loss of training set and validation set. Specifically, given the network prediction of both training and validation set, it calculates the loss and inserts it into the respective list.

Parameters

`y_pred_train`: `np.ndarray`
The network prediction on training set (i.e the forward propagation's result applied on the training set). It has a shape of (c, N) where c is the number of target classes and N is the number of samples.

`y_train`: `np.ndarray`
The one-hot encoded matrix which contains the true labels. It represents the labels of training set. It has the same shape of `y_pred_train`.

`y_pred_val`: `np.ndarray`)
The network prediction on validation set (i.e the forward propagation's result applied on the validation set). It has a shape of (c, N) where c is the number of target classes and N is the number of samples.

`y_val`: `np.ndarray`
The one-hot encoded matrix which contains the true labels. It represents the labels of validation set. It has the same shape of `y_pred_val`.

`train_loss_list`: `List[float]`
A list where each element represents the loss on the training set of a certain epoch.

`val_loss_list`: `List[float]`
A list where each element represents the loss on the validation set of a certain epoch.

Returns

`train_loss_list`: `List[float]`
The updated list where the last element is the loss on training set of the current epoch.

`val_loss_list`: `List[float]`
The updated list where the last element is the loss on validation set of the current epoch.

2.8.3 Algorithms implemented

In this section the major implemented algorithms will be shown. That is, all the methods required for training (forward propagation, backpropagation, fit,

etc.).

Forward propagation

```
def __forward_propagation(self, x):
    """
    The function apply the forward propagation.

    Parameters:
    ----------
    x (matrix): Input matrix of shape (d, N) where 'd' is the number of↩
     feature and N is the number of samples.

    Returns:
    ---------
    z (matrix): Output matrix of shape (c, N) where 'c' is the number ↩
    of classes and N is the number of samples.
    """

    # copy input
    z = x

    for l in range(self.n_layers):
        a = np.matmul(self.weights[l], z) + self.biases[l]
        z = self.activation_list[l](a)
    return z
```

The goal of the forward propagation is to compute the output of the network given a certain input. The algorithm computes:

$$z^{(l)} = f(a^{(l)}) \text{ where } a^{(l)} = W^{(l)} z^{(l-1)} + b^{(l)}$$

where:

- $z^{(l)}$ is the output of the current layer l
- f is the activation function
- a^l is the input of the current layer l
- W^l is the matrix of weights of the current layer
- $z^{(l-1)}$ is the activation of previous layer. For the first hidden layer is equal to the input.
- b^l is the vector of biases of the current layer

Basically, for each layer, the function computes the input a^l and then applies the activation function f to that input, obtaining z^l.

Note that is a private method since it should be used only in the `NeuralNetwork` class.

Parameters

x: `np.ndarray`
A matrix with a shape of (d, N) (where d is the number of features and N is the number of samples).

Returns

z: `np.ndarray`
A matrix with a shape of (c, N) (where c is the number of target classes).

Forward propagation (training)

```
def __forward_propagation_training(self, x):
    """
    The function apply the forward propagation (only for training).

    Parameters:
    ----------
        x (matrix): Input matrix of shape (d, N) where 'd' is the ↩
    number of feature and N is the number of samples.

    Returns:
    ---------
        z_layer (list): List of output matrix for each layer,
        which means that each z.shape is (m, N) where m is number of ↩
    neuron and N number of samples.
        z_derivative (list): A list of the derivatives of activation ↩
    funcions computed into the input of each layer.
    """

    # init the lists
    z_layer = []
    z_derivative = []

    # First layer contains data points.
    # Data point can't change (but are in fact used to train the ↩
    network by adjusting the weights and biases),
    # this layer can be considered as an activation layer. Therefore ↩
    append input to z_layer
    z_layer.append(x)

    for l in range(self.n_layers):
        a = np.matmul(self.weights[l], z_layer[l]) + self.biases[l]
        g = self.activation_list[l](a)
        g_der = self.activation_list[l](a, 1)

        z_layer.append(g)
        z_derivative.append(g_der)

    return z_layer, z_derivative
```

The `forward_propagation` function computes only the activation values of the output layer. In order to perform training, the backpropagation (see 2.8.3) algorithm needs to use the derivative of the activation function for each layer.

Therefore, the function above computes the output and its derivative for each layer.

Parameters

`x`: `np.ndarray`
A matrix with a shape of (d, N) (where d is the number of features and N is the number of samples).

Returns

`z_layer`: `List[np.ndarray]`
A list that contains th output of each layer (including the input), thus z^l.

`z_layer_derivative`: `List[np.ndarray]`
A list that contains the derivatives of the output of each layer, hence a list $[f_1'(a^{l_1}), f_2'(a^{l_2}), ..., g'(a^{l_n})]$ where $f_1'(a^{l_1})$ is the derivative of activation function of the first layer, $f_2'(a^{l_2})$ is the derivative of activation function of the second layer, and so on, up to $g'(a^{l_n})$ which is the derivative of activation function of the last layer l_n.

Back-propagation

```
def back_propagation(self, x, target):
    z_layer, z_derivative = self.__forward_propagation_training(x)
    delta_list = self.__compute_delta(z_layer, z_derivative, target)
    weights_deriv, biases_deriv = self.__compute_derivatives(delta_list
    , z_layer)

    return weights_deriv, biases_deriv
```

Backpropagation is an algorithm that is limited to the calculation of gradients, specifically the derivatives of the error function with respect to the weights (resp. biases) of the network. Once the gradients are calculated, the gradient descent algorithm uses them to update the weights of the network.

The back-propagation algorithm is as follows:

Algorithm 1: Back-propagation Algorithm

1 Do $Forward\_Propagation$ in order to compute a_i^n and z_i^n
2 `Compute` δ`:`
3 $\delta_k^n = g'(a_k^n)\frac{\partial E^n}{\partial y_k^n}$ ▷ δ output nodes
4 $\delta_h^n = f'(a_h^n)\sum_k w_{kh}\delta_k^n$ ▷ δ hidden nodes
5 $\forall w_{ij}$, compute $\frac{\partial E^n}{\partial w_{ij}} = \delta_i^n \cdot z_j^n$ ▷ if j "runs" on input nodes $z_j^n = x_j^n$

Therefore, based on the algorithm above, 4 ingredients are needed for back-propagation:

1. the derivative of activation functions of each layer (hidden and output);
2. the partial derivative of loss function with respect to the prediction for the output δ
3. the δ of output and hidden nodes
4. the activation function of each layer

Firstly, the derivative of activation functions are needed. Therefore, by applying the `forward_propagation_training` algorithm, a list of derivative is obtained. The list has the form $[f_1'(a^{l_1}), f_2'(a^{l_2}), ..., g'(a^{l_n})]$.

Later, the `compute_delta`, as the name suggests, computes the delta of output and hidden nodes.

Parameters

`x:`	`np.ndarray` The training set organized as matrix (d, N) where d is number of features and N is the number of samples.
`target:`	`np.ndarray` The one-hot encoded matrix of labels.

Returns

`weights_deriv:`	`List[np.ndarray]` The partial derivatives of weights
`biases_deriv:`	`List[np.ndarray]` The partial derivatives of biases

How to compute the deltas

```
def __compute_delta(self, z_layer, z_derivative, target):
    """
    Compute delta.
    Starting from the last layer:
        1) Compute the delta of the output layer and insert it into the↩
     list (pos 0)
        2) Compute the delta of hidden(s) layer and insert it into the ↩
    list (first position)
            - Therefore, if there were more hidden layers, we would ↩
    have an ordered list of deltas.
    """
    delta = []

    # output layer
    z_last_layer = z_layer[-1]
    cost_function_der = self.error_function(z_last_layer, target, 1)
    der_output = z_derivative[-1]
    delta.append(cost_function_der * der_output)

    # hidden layers
    for l in range(self.n_layers - 1, 0, -1):
        # w^(l3)^T * delta^(l3) * sigmoide_der(z^l2)
```

```
            curr_delta = z_derivative[l - 1] *
                         np.matmul(self.weights[l].transpose(), delta[0])
            delta.insert(0, curr_delta)

        return delta
```

The second step of the back-propagation algorithm is to compute the partial derivative of loss function ($\frac{\partial E^n}{\partial y_k^n}$). This is needed in order to compute the δ of the output layer. Once this value is computed, the function proceeds to compute the deltas:

- for the output, δ, we simply take the last element from the list of derivatives, which is $g'(a^{l_n})$, and then multiply it by the partial derivative of loss function. Then, this value is inserted into a list (at position 0).
- the δ of hidden nodes is calculated as follows:

$$\delta^l = ((W^{(l+1)})^T \delta^{l+1}) f'(a^l)$$

  where:

  - δ^l is a matrix where each column corresponds to the errors related to a single sample;
  - W^{l+1} is the weight matrix that connects layer l to the subsequent layer $l+1$;
  - δ^{l+1} is the delta of the next layer;
  - $f'(a^l)$ is the derivative of the activation function calculated on the input of layer l.

  Similarly, in Python, we have:

  - `self.weights[l]` corresponds to W^{l+1}. This is true because `self.weights` $= [w_1, w_2, ..., w_n]$ is a list of weight matrices, thus starting from index 0.
  - `z_derivative[l - 1]` corresponds to $f'(a^l)$ for the same reason (i.e., it is a list that starts from index 0).
  - `delta[0]` corresponds to δ^{l+1}. The delta of the next layer is always inserted at the front of the list (i.e., at position 0). Therefore, in the end, there will be a list $[\delta^{l_1}, \delta^{l_2}, ..., \delta^{l_n}]$, where the first element is the delta associated with the first hidden layer, the second with the second hidden layer, and so on, until the last one which represents the delta associated with the output layer.
  - Hence, we iterate backwards starting from `self.n_layers - 1`, which is the number of hidden layers (the output layer has already been calculated), and propagate the error backward by calculating the delta of the current layer and inserting it at the front of the list.

Parameters

`z_layer`: `List[np.ndarray]`
A list where each element represents the output of corresponding layer. The first element is the input matrix (i.e the training set with a shape of (d, N))

`z_derivative`: `List[np.ndarray]`
A list where each element represents the derivative of activation function of the output of corresponding layer.

`target`: `List[np.ndarray]`
The one-hot encoded matrix (with a shape of (c, N)) that will be used to compute the partial derivative of loss function with respect to the prediction (i.e $\frac{\partial E}{\partial y}$).

Returns

`deltas`: `List[np.ndarray]`
A list of deltas $[\delta^{l_1}, ..., \delta^{l_n}]$.

Partial derivative

```
def __compute_derivatives(self, delta, z):
    """
    This function represents the last step of the back-propagation ↩
    algorithm.
    It computes the partial derivatives: delta_i * z_j,
    where delta_i is the of the node 'i' and z_j is the output of node ↩
    'j'.

    Parameters:
    ----------
        delta (list): the list of the the errors of each layer.
        z (list): the list of matrices where each element represents ↩
    the output of each layer.

    Returns:
    ---------
        weights_deriv (list): a list of derivatives of weights
        biases_deriv (list): a list of derivatives of biases
    """
    weights_deriv=[]
    biase_deriv=[]

    for l in range(self.n_layers):
        der_c = np.matmul(delta[l], z[l].transpose())
        weights_deriv.append(der_c)
        biase_deriv.append(np.sum(delta[l], 1, keepdims=True))

    return weights_deriv, biase_deriv
```

The last step is the computing of partial derivatives for each w_{ij}. Therefore, the goal is to compute $\frac{\partial E^n}{\partial w_{ij}} = \delta_i^n \cdot z_j^n$.

Parameters

`delta:`	`List[np.ndarray]` The list of delta calculated by the `__compute_delta` function.
`z:`	`List[np.ndarray]` A list where each element represents the output of corresponding layer. The first element is the input matrix (i.e the training set with a shape of (d, N)).

Returns

`weights_deriv:`	`List[np.ndarray]` The partial derivatives of weights.
`biases_deriv:`	`List[np.ndarray]` The partial derivatives of biases.

Gradient descent

```
def __gradient_descent(self, learning_rate, weights_derivative, ↩
    biases_derivative, momentum = 0.0):
    """
    The function apply the gradient descent.
    When momentum = 0.0 update rule is: w = w - learning_rate * g where↩
     'g' is the gradient (i.e weights and bias derivative).
    When momentum >= 0.0 update rule is:
        i) velocity = momentum * velocity - learning_rate * g
        ii) w = w + velocity

    Parameters:
    ---------
        learning_rate (float):
        weights_derivative (list):
        biases_derivative (float):
        momentum (float): float hyperparameter >= 0 that accelerates
        gradient descent in the relevant direction and dampens
        oscillations.
        0 is vanilla gradient descent. Defaults to 0.0.
    """

    if momentum == 0:
        # standard gradient descent
        for i in range(self.n_layers):
            self.weights[i] = self.weights[i] - (learning_rate *
                weights_derivative[i])
            self.biases[i] = self.biases[i] - (learning_rate *
                biases_derivative[i])
    else:
```

```
            for i in range(self.n_layers):
                self.velocity_w[i] = (momentum * self.velocity_w[i]) -
                    (learning_rate * weights_derivative[i])
                self.velocity_b[i] = (momentum * self.velocity_b[i]) -
                    (learning_rate * biases_derivative[i])
                self.weights[i] += self.velocity_w[i]
                self.biases[i] += self.velocity_b[i]
```

The function above computes the update rule based on the gradient calculated by the backpropagation function. Therefore, the Python function defines:

- the vanilla gradient descent (if $momentum = 0$), which is done applying the rule:
$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$
where $0 < \eta < 1$ is hyper-parameter.

- the gradient descent with momentum, otherwise.
$$v_t = \gamma v_{t-1} + \eta \Delta w_t$$
$$w_{t+1} = w_t - v_t$$
where $0 < \alpha < 1$ is called **momentum coefficient** and t is the step.

Parameters

`learning_rate`: `float`
The η used for the update rule.

`momentum`: `float`
loat hyperparameter ≥ 0 that accelerates gradient descent in the relevant direction and dampens oscillations. 0 is vanilla gradient descent.

`weights_deriv`: `List[np.ndarray]`
The partial derivatives of weights calculated during backpropagation.

`biases_deriv`: `List[np.ndarray]`
The partial derivatives of biases calculated during backpropagation.

Train

In the context of neural networks, the training and validation sets are subjected to an iterative process.

The *learning process* can occur through three strategies:

- **batch**: the derivative of the error function is calculated with respect to the parameters by considering all the data in the training set, and then, the network is updated; all within an epoch.

- **online**: the derivative of the error function is computed for each element of the training set individually and the network is updated accordingly.

- **mini-batch**: It divides the training set into blocks and treats each block as if it were a single piece of data. So, the network is updated at the end of each block

Furthermore, it's possible to define the **early stop** criterion, which mean there is a condition (e.g *patience*) that stops the learning process after a certain number of epochs.

In general, the training algorithm can be viewed as follows:

Algorithm 2: Train Algorithm

Input: x_train, y_train, x_val, y_val,
$epochs$,
$learning\_rate = 0.01$,
$momentum = 0.0$,
$mode =$ 'batch',
$num\_mini\_batches = 32$,
$early\_stopping\_criteria$

validate inputs
$e_{min} \leftarrow 0$;
for $epoch = 1$ **to** $epochs$ **do**
  if $mode =$ *'batch'* **then**
    perform forward-prop and compute gradients of all training set
    update parameters with full dataset
  end if
  else if $mode =$ *'mini-batch'* **then**
    divide dataset into $num\_mini\_batches$;
    for *each mini-batch* **do**
      perform forward-prop and compute gradients
      update parameters for current mini-batch;
    end for
  end if
  else if $mode =$ *'online'* **then**
    for *each sample* **do**
      perform forward-prop and compute gradient
      update parameters for current sample;
    end for
  end if
  $e_{val}, e_{train} \leftarrow$ compute validation/training error;
  $a_{val}, a_{train} \leftarrow$ compute validation/training accuracy;
  if $e_{val} < e_{min}$ **then**
    $best\_network \leftarrow$ copy current network;
    reset patience;
  end if
  $res = handle\_early\_stopping\_criteria()$
  if res **then**
    stop learning;
  end if
end for
return $best\_network$, $statistics$

Therefore, based on the value of $mode$ the algorithm computes the gradients and update the network accordingly. After that, it proceeds to calculate the error on both training set and validation set. If the validation error decreased, the current network is copied in order to return the network which have the minimum validation error. Next, it checks whether learning should be stopped or not depending on the early stopping condition. Finally, the algorithm returns the network which have the minimum validation error and several statistics such as a list of errors for both training and validation, number of epochs, time to training, etc...

That said, the implementation of the above algorithm in Python is shown:

```
def train(self,
        x_train: np.ndarray,
        y_train: np.ndarray,
        x_val: np.ndarray,
        y_val: np.ndarray,
        epochs: int,
        early_stopper: EarlyStopping,
        learning_rate = 0.001,
        momentum = 0.0,
        mode='batch',
        num_mini_batches = 32,
        f1_avg_type = None):

    # check if all inputs are valid
    self.__train_validation(x_train, y_train, x_val, y_val, epochs,
                            learning_rate, momentum, mode, ↩
    num_mini_batches)
    # in order to return the highest performing model, which is not ↩
    necessarily the one of the last epoch
    best_net, min_error = None, None
    epoch_best_net = epochs
    # how much time was spent on learning
    tot_time = 0
    # list containing training loss, f1-score and accuracy of each ↩
    epoch
    train_loss_list, train_ac_list, train_f1_list = [], [], []
    # list containing validation loss and training accuracy of each ↩
    epoch
    val_loss_list, val_ac_list, val_f1_list = [], [], []
    # for early stopping
    stopped_epoch, flag = None, False

    print(f"Learning mode is {mode}\n[==============================]")

    for e in range(epochs):
        start_time = perf_counter()
        # do batch - online - minibatch based on 'mode'
        self.__learning(x_train, y_train, learning_rate, momentum,
                        mode, num_mini_batches)
        end_time = perf_counter() - start_time
        tot_time += end_time
        # get network prediction (validation can be none if network
```

```
            is trained using all data)
            y_pred_train = self.__forward_propagation(x_train)
            y_pred_val = self.__forward_propagation(x_val) if x_val is
                                                           not None else ↩
        None
            # updating lists
            train_loss_list, val_loss_list = self.__get_train_val_loss(
                                               y_pred_train, y_train,
                                               y_pred_val, y_val,
                                               train_loss_list,
                                               val_loss_list)
            train_ac_list, val_ac_list = self.__get_train_val_accuracy(
                                           y_pred_train, y_train,
                                           y_pred_val, y_val,
                                           train_ac_list, val_ac_list)
            train_f1_list, val_f1_list = self.__get_train_val_f1_score(
                                           y_pred_train, y_train,
                                           y_pred_val, y_val,
                                           train_f1_list, val_f1_list,
                                           f1_avg_type)

            print(f"Epoch {e + 1}/{epochs}")
            print(f"[==============================] -
                    {end_time:.2f}s/step -",
                  f"loss: {train_loss_list[e]:.4f} -
                    accuracy: {train_ac_list[e]:.4f} -",
                 (f"val_loss: {val_loss_list[e]:.4f} -
                    val_accuracy: {val_ac_list[e]:.4f} - "
                    if x_val is not None else ""),
                  f"p: {early_stopper.get_patience_counter()}", end='\n')

            # for best network (min validation error)
            if x_val is not None and (best_net is None or
                                      val_loss_list[e] < min_error):
                min_error = val_loss_list[e]
                best_net = self.copy_network()
                # the epoch of min validation error
                epoch_best_net = e
                early_stopper.reset_patience_counter()
                flag = False
            else:
                flag = True

            # early stopping
            if x_val is not None:
                # in order to verify the early stop condition
                e_train = train_loss_list[-1]
                e_val_curr = val_loss_list[-1] if x_val is not None
                                               else None
                # if condition of early stopping is verfied stop learning
                if early_stopper.check_early_stop_condition(e, e_train,
                                    e_val_curr, min_error, flag):
                    print(f"Early stopping condition met at epoch {e}.")
                    stopped_epoch = e
```

```
                break

        # update best network
        if best_net != None:
            self = copy.deepcopy(best_net)

        print(f"Total time: {tot_time:.2f}s")

        report = {
            "stop": stopped_epoch,
            "epoch_best": epoch_best_net,
            "Accuracy_train": train_ac_list,
            "Accuracy_val": val_ac_list,
            "Loss_train": train_loss_list,
            "Loss_val": val_loss_list,
            "f1-score_t": train_f1_list,
            "f1-score_v": val_f1_list,
            "Time": f"{tot_time:.2f}"
        }

        return report
```

Note that it's not necessary to returns a new object ($best\_net$) since there is a `deepcopy` function, which constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Furthermore, there are several controls on the validation set (whether it is none or not). This is because once the model is found in the selection and validation phase, one might want to train the model on the whole training set.

Finally, in order to make the code more readable and modular, the training function was broken into several parts. Specifically, the **learning** function is further divided into several parts: depending on the value of the $mode$, one of the possible learning strategies will be called (i.e `__batch_learning`, `__online_learning`, `__mini_batch_learning`).

Parameters

`x_train`:	`np.ndarray` A matrix with a shape of (d, N) where d is the number of features and N is the number of samples. It represents the training set.
`y_train`:	`np.ndarray` The one-hot encoded matrix which contains the true labels. It represents the labels of training set.
`x_val`:	`np.ndarray` A matrix with a shape of (d, N) where d is the number of features and N is the number of samples. It represents the validation set.
`y_val`:	`np.ndarray` The one-hot encoded matrix which contains the true labels. It represents the labels of validation set.

`epochs`:	`int`): An integer that represents how many times repeat the learning process.
`learning_rate`:	`float`, *Default=0.01* The eta value $0 < \eta < 1$ used for the update rule.
`momentum`:	`float`, *Default=0.0* A float hyperparameter ≥ 0 that accelerates gradient descent in the relevant direction and dampens oscillations. 0 is vanilla gradient descent.
`mode`:	`str`, *Default='batch'* A string that represents the type of learning. Therefore, it can assume three possible values ($online$, $batch$ or $mini - batch$). If $mode \neq' online'$ or $mode \neq' batch'$ the $mini - batch$ learning will be applied. Defaults to $batch$.
`num_mini_batches`:	(`int`), *Default=32* An integer that represents the number of mini-batches. If $mode \neq' mini - batch$ it can assume any value.
`early_stopper`	(`EarlyStopping`) The *early-stop* condition.

Returns

`report` (`dict`)
A dictionary that contains:
`stop` (`int`): the epoch at which learning process was interrupted.
`epoch_best` (`int`): the last epoch when validation error was the minimum (i.e the best net).
`time` (`float`): a float that represents the time needed to learning.
`train_loss_list` (`List[float]`): a list where each element represents the training set loss for each epoch.
`val_loss_list` (`List[float]`): a list where each element represents the validation set loss for each epoch.
`train_ac_list` (`List[float]`): a list where each element represents the training set accuracy for each epoch.
`val_ac_list` (`List[float]`): a list where each element represents the validation set accuracy for each epoch.
`f1_score_t` (`List[float]`): a list where each element represents the training set f1-score for each epoch.
`f1_score_v` (`List[float]`): a list where each element represents the validation set f1-score for each epoch.

Returns

Raise `ValueError` if parameters are not correct.

Learning

```
def __learning(self, x_train, y_train, learning_rate, momentum, mode,
               num_mini_batches):
    if mode == 'batch':
        return self.__batch_learning(x_train, y_train,
                                     learning_rate, momentum)
    elif mode == 'online':
        return self.__online_learning(x_train, y_train,
                                      learning_rate, momentum)
    else:
        return self.__mini_batch_learning(x_train, y_train,
                                          learning_rate, momentum,
                                          num_mini_batches)
```

The `__learning` function takes as input the same parameters as the train function, except for the `epoches`, the `patience` and validation set parameters. Similarly, each learning functions will also take the same parameters as input, except for the `__online_learning` and `__batch_learning` functions that do not require the number of mini-batches. Furthermore, the `__online_learning` and `__mini_batch_learning` shuffle the data at each epoch, thereby preventing the network from learning the order of the elements.

Note that all the learning functions are defined as private, following good programming practices (encapsulation), where implementation details are hidden. This ensures that they cannot be accessed or modified from outside the class, helping to avoid errors and make the code more modular. As a result, the client only interacts with the `train` function without needing to worry about the internal details of how the learning algorithm is implemented.

```
def __batch_learning(self, x_train, y_train, learning_rate, momentum):
    weights_deriv, bias_deriv = self.back_propagation(x_train, y_train)
    self.__gradient_descent(learning_rate, weights_deriv,
                            bias_deriv, momentum)
```

```
def __online_learning(self, x_train, y_train, learning_rate, momentum):
    # get number of samples
    N = x_train.shape[1]
    # create a list of samples
    SAMPLES = [[i] for i in np.arange(N)]
    # shuffle them
    SAMPLES = np.random.permutation(SAMPLES)
    for sample in SAMPLES:
        x_sample = x_train[:, sample].reshape(-1, 1)
        y_sample = y_train[:, sample].reshape(-1, 1)

        weights_deriv, bias_deriv = self.back_propagation(x_sample,
                                                          y_sample)
        self.__gradient_descent(learning_rate, weights_deriv,
                                bias_deriv, momentum)
```

```
def __mini_batch_learning(self, x_train, y_train,
```

```
                            learning_rate, momentum, num_mini_batches):
        n_batches = Utility.split_batch(y_train, num_mini_batches)
        for batch in n_batches:
            x_batch = x_train[:, batch]
            y_batch = y_train[:, batch]

            # get partial derivative
            weights_deriv, bias_deriv = self.back_propagation(x_batch,
                                                              y_batch)

            # update weights and biases
            self.__gradient_descent(learning_rate, weights_deriv,
                                    bias_deriv, momentum)
```

2.8.4 Summary

The following table contains a summary of `NeuralNetwork` methods.

Method	Visibility	Parameters	Description

`__init__`	`public`	`input_size: int`
		`output_size: int`
		`n_hidden_size: int[]`
		`m_neurons_list: int`
		`activation_list: ActivationFunction[]`
		`error_function: LossFunction`
		`weights_list: List[np.ndarray]`
		`biases_list: List[np.ndarray]`
			Initialize the network object. The parameters define the structure and behavior of the neural network.

`summary`	`public`		Print the network structure in a tabular format.

`train`	`public`	`x_train: np.ndarray`
		`y_train: np.ndarray`
		`x_val: np.ndarray`
		`y_val: np.ndarray`
		`epochs:int`
		`learning_rate:float`
		`momentum:float`
		`mode:string`
		`num_mini_batches:int`
		`early_stopper: EarlyStopping`
			Trains the model for a fixed number of epochs (dataset iterations).

Method	Visibility	Parameters	Description
`predict`	`public`	`input`: `np.ndarray`	Predict a single element. Thus, the input has a shape of $(d, 1)$ where d is the number of features. Finally, it returns a vector of probabilities.

`evaluate_model`	`public`	`test`: `np.ndarray`
		`target`: `np.ndarray`
		`target_names`: `List[str]`
			Compute how accurate is the model on the test set. It returns the accuracy, a confusion matrix and a report (F1-score, precision and recall)

`save_model`	`public`	`file_path`: `String`	Save the trained model.
`load_model`	`public`	`file_path`: `String`	Load the trained model and returns it.
`forward_propagation`	`private`	`x`: `np.ndarray`	It computes the output of the network given the input `x`. Then, it returns that output, which has a shape of (c, N) where c is the number of target classes and N is the number of samples.
`forward_propagation_training`	`private`	`x`: `np.ndarray`	It computes the output of each layer and inserts it into a list (the input `x` is included in that list). At the same time it computes the derivative of the output of each layer and inserts it into another list. Then, it returns both lists. The first one ha lengths `n_layers + 1` (because of the input) and `n_layers` for the second one.

Method	Visibility	Parameters	Description
`back_propagation`	`private`	`x: np.ndarray` `target: np.ndarray`	It computes the back-propagation algorithm, which means that it applies forward propagation, then calculates the deltas and after that it computes partial derivatives for weights and biases. Finally, it returns a list of partial derivatives of the weights and biases.

`compute_delta`	`private`	`z_layer: List[np.ndarray]`
		`z_layer_der: List[np.ndarray]`
		`target: np.ndarray`
			It calculates the deltas of output and hidden layers. Finally it returns a list of deltas where the first position is the δ of first layer, the second one is the second layer and so on.

`compute_derivatives`	`private`	`delta: List[np.ndarray]` `z_layer: List[np.ndarray]`	It computes $\frac{\partial E}{\partial w_{ij}} = \delta_i \cdot z_j$. It returns the list of derivatives of weights and biases

`gradient_descent`	`private`	`learning_rate: float`
		`weights_derivative: List[np.ndarray]`
		`biases_derivative: List[np.ndarray]`
		`momentum: float`
			It applies the gradient descent. If $momentum = 0$ then it applies the vanilla gradient descent, otherwise it use the momentum for update rule.

`learning`	`private`	`x_train: np.ndarray`
		`y_train: np.ndarray`
		`learning_rate:float`
		`momentum:float`
		`mode:string`
		`num_mini_batches:int`
			Based on $mode$'s value the function calls the appropriate learning function ($online$, $batch$, etc).

Method	Visibility	Parameters	Description
batch _learning	private	x_train: np.ndarray y_train: np.ndarray learning_rate:float momentum:float	It calculates the derivative of the error function with respect to the parameters by considering all the data points in the training set. Thus, it applies the gradient descent.
online _learning	private	x_train: np.ndarray y_train: np.ndarray learning_rate:float momentum:float	It calculates the derivative of the error function with respect to the parameters by considering each individual data point of the training set (shuffled). Thus, it applies the gradient descent using those derivatives. This is repeated for all elements of the training set.
mini_batch _learning	private	x_train: np.ndarray y_train: np.ndarray learning_rate:float momentum: float num_mini_batches:int	Firstly it divides the training set in num_mini_batches, then it calculates the derivative of the error function with respect to the parameters by considering each batch. Thus, it applies the gradient descent using those derivatives. This is repeated for all batches.

Method	Visibility	Parameters	Description

`train_val_accuracy`	`private`	`y_pred_train`: `np.ndarray`
		`y_train`: `np.ndarray`
		`y_pred_val`: `np.ndarray`
		`y_val`: `np.ndarray`
		`train_ac_list`: `List[float]`
		`val_ac_list`: `List[float]`
			It computes the accuracy of training set and validation set. Specifically, given the network prediction of both training and validation set, it calculates the accuracy and inserts into the respective list. Finally it returns both lists.

`train_val_loss`	`private`	`y_pred_train`: `np.ndarray`
		`y_train`: `np.ndarray`
		`y_pred_val`: `np.ndarray`
		`y_val`: `np.ndarray`
		`train_loss_list`: `List[float]`
		`val_loss_list`: `List[float]`
			It computes the loss of training set and validation set. Specifically, given the network prediction of both training and validation set, it calculates the loss and inserts into the respective list. Finally it returns both lists.

`compute_accuracy`	`private`	`y_net`: `np.ndarray` `target`: `np.ndarray`	It returns the fraction of correctly classified samples (float)
`train_validation`	`public`	Same inputs of `train` function	Raise a `ValueError` if any input (of train function) is not of the right type or does not have legal values.

Table 2.14: `NeuralNetwork` class.

Chapter 3

Part B

This chapter provides an introduction to the MNIST dataset and explores the process of training a neural network using the images from the dataset as input.

3.1 Introduction

The MNIST database of handwritten digits has a training set of 60.000 examples, and a test set of 10.000 examples. Each example is an image of a digit (0 to 9). The images are normalized to fit into a 28x28=784 pixel bounding box and anti-aliased, introducing grayscale levels. Therefore, the input of the network is 784 and the output is 10.

The aim is to study the learning of the neural network, specifically the number of epochs required, the trend of the error on the validation set, etc., as the number of nodes in a single hidden layer and the hyperparameters learning rate (eta) and momentum vary.

Hence, we have a *shallow neural network* with 784 of input neurons and 10 output neurons (as shown in Figure 3.2).

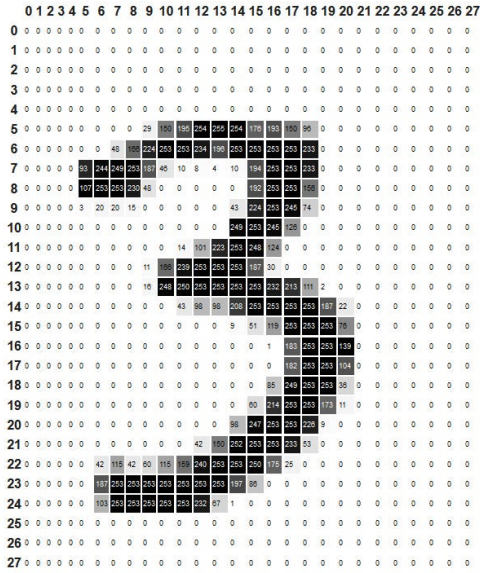


Figure 3.1: Example of an image from MNIST dataset

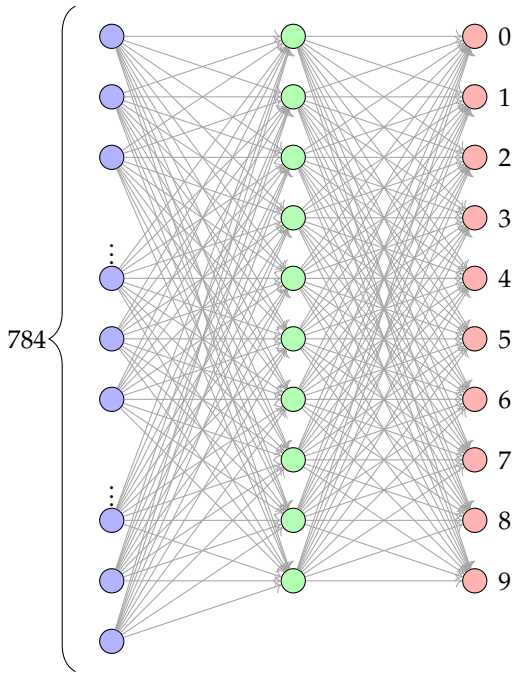


Figure 3.2: Neural Network for MNIST dataset

3.2 Experimental setup

The aim of this project is to study the learning process of a neural network by varying the learning rate η and the momentum α for at least 5 different dimensions of the hidden layer (i.e., by changing the number of neurons). The goal is to find the optimal hyperparameters to train the network. In order to do this, the model selection was performed using the **Grid-Search** method and the validation was carried out using a **Hold-out** set. The metric used to evaluate the model is the **accuracy**.

For each hyperparameter, the network had the following architecture:

- **Number of input nodes**: 784
- **Number of hidden layers**: 1
- **Activation function of hidden layer**: Sigmoid function.
- **Number of output nodes**: 10
- **Activation function of output layer**: Identity function. This was done in order to use cross-entropy with softmax as error function.
- **Error function**: Cross-entropy with softmax.
- **Update rule**: Gradient descent with momentum α.

- **Patience**: 15, since it is a value that is not too low and not too high in this context, so as not prematurely stop the training, especially at the beginning where the model is very inaccurate.
- **Learning process**: batch.
- **Training set**: 48.000 samples.
- **Validation set**: 12.000 samples $(20\%$ of 60.000 samples$)$.
- **Test set**: 10.000 samples.
- **Validation mode**: Hold-out. Note that, as a usual rule, k-fold cross validation is recommended for grid search (or random forest). However, for simplicity and also because of the very large number of models to be evaluated, a classical approach was chosen in order not to overly stretch the execution time.
- **Maximum number of epochs defined**: 500.

The hyperparameter to evaluate:

- **Number of neurons**: $[20, 50, 100, 200, 500]$
- **Learning rate** η: $[0.0002, 0.0001, 0.00009, 0.00005]$
- **Momentum** α: $[0.5, 0.75, 0.9]$

Therefore, the number of combinations to test is $5 \times 4 \times 3 = 60$.

3.3 Results

Before seeing the results, the following should be kept in mind:

- The "**epochs**" column stands for the number of epochs that were needed to obtain the network with minimum validation error. Therefore, since there is an early stop condition (i.e $patience = 15$), the epoch at which learning was stopped is given by $epochs + patience$ (e.g if epochs is 134, learning was stopped at epoch number 149).
- Time values are expressed in seconds.
- The model which have the higher accuracy and lower loss (on validation set) are highlighted in green.
- The model which have an accuarcy lower than 50% are highlighed in red.
- The model which have an accuracy higher than 60% and lower than 80% are highlighted in violet.

- The model which have an accuracy higher than 80% and lower than 90% are highlighted in blue.

ID	Neurons	η	α	A_{train}	A_{val}	$Loss_{train}$	$Loss_{val}$	Epochs	Time
0	0,0002	0,5	20	0,7600	0,7621	33048,6463	8363,2833	27	11,
1	0,0002	0,75	20	0,8365	0,8336	25465,1957	6636,2394	59	23,2
2	0,0002	0,9	20	0,8952	0,8834	18965,1326	5286,5035	152	49,9
3	0,0001	0,5	20	0,9578	0,9488	7267,9212	2231,3016	224	80,8
4	0,0001	0,75	20	0,9793	0,9563	3774,1476	1866,2265	471	146,9
5	0,0001	0,9	20	0,9758	0,9488	4252,3218	2211,1583	264	83,1
6	0,00009	0,5	20	0,9679	0,9498	5494,8852	2045,4467	499	148,
7	0,00009	0,75	20	0,9760	0,9532	4290,7121	2003,3445	413	124,2
8	0,00009	0,9	20	0,9782	0,9528	3963,5225	2013,6344	292	92,0
9	0,00005	0,5	20	0,9574	0,9458	7164,0327	2229,6539	499	179,6
10	0,00005	0,75	20	0,9692	0,9518	5232,2216	1980,0888	499	155,2
11	0,00005	0,9	20	0,9749	0,9516	4272,6237	2045,1257	386	114,9
12	0,0002	0,5	50	0,3098	0,3033	87379,7448	21926,3722	67	45,3
13	0,0002	0,75	50	0,4101	0,4026	65695,7808	16561,7625	41	32,83
14	0,0002	0,9	50	0,3226	0,3157	74952,2505	19138,8584	440	249,81
15	0,0001	0,5	50	0,9543	0,9432	7665,6490	2321,1539	241	141,21
16	0,0001	0,75	50	0,9260	0,9193	12442,4011	3305,1932	69	47,94
17	0,0001	0,9	50	0,9463	0,9296	8974,8230	3063,2725	204	119,28
18	0,00009	0,5	50	0,9741	0,9568	4599,1414	1747,5306	499	272,42
19	0,00009	0,75	50	0,9621	0,9455	6385,0896	2241,2947	256	150,05
20	0,00009	0,9	50	0,9101	0,8983	16309,5083	4512,5859	114	70,93
21	0,00005	0,5	50	0,9709	0,9588	5094,2846	1669,4626	499	274,8
22	0,00005	0,75	50	0,9793	0,9603	3783,4681	1610,6280	499	271,38
23	0,00005	0,9	50	0,9936	0,9684	1587,0031	1323,8896	438	245,5
24	0,0002	0,5	100	0,3867	0,3785	71658,6916	17981,2579	95	97,53
25	0,0002	0,75	100	0,5079	0,5103	55138,7388	14047,0703	71	78,03
26	0,0002	0,9	100	0,1124	0,1123	110476,8010	27621,8203	74	84,56
27	0,0001	0,5	100	0,9512	0,9426	8346,7292	2469,6772	245	228,44
28	0,0001	0,75	100	0,9151	0,9090	15006,9701	3968,5506	60	66,45
29	0,0001	0,9	100	0,8947	0,8866	18597,4930	5184,0496	81	85,31
30	0,00009	0,5	100	0,8898	0,8832	19068,7259	4977,1783	94	96,94
31	0,00009	0,75	100	0,9429	0,9308	9759,6434	2882,0581	136	136,28
32	0,00009	0,9	100	0,8658	0,8580	21512,0606	5794,2731	125	146,14
33	0,00005	0,5	100	0,9700	0,9605	5207,2099	1664,2554	499	454,01
34	0,00005	0,75	100	0,9700	0,9549	5163,0176	1934,1264	499	438,67
35	0,00005	0,9	100	0,9726	0,9473	4765,1409	2217,6891	481	439,47
36	0,0002	0,5	200	0,4163	0,4188	73265,5961	18360,1046	20	53,46
37	0,0002	0,75	200	0,7126	0,7067	37921,4174	9749,3634	73	129,56
38	0,0002	0,9	200	0,0974	0,0978	173031,9304	43045,2218	9	37,61
39	0,0001	0,5	200	0,9549	0,9481	7573,2287	2278,7497	222	343,73
40	0,0001	0,75	200	0,5065	0,5013	61289,0152	15423,5668	98	166,97
41	0,0001	0,9	200	0,8275	0,8115	27146,2096	7356,7609	98	167,34

ID	Neurons	η	α	A_{train}	A_{val}	$Loss_{train}$	$Loss_{val}$	Epochs	Time
42	0,00009	0,5	200	0,9509	0,9430	8235,4989	2413,2246	251	390,38
43	0,00009	0,75	200	0,8105	0,8039	30365,1114	7774,8464	94	160,86
44	0,00009	0,9	200	0,5773	0,5727	46479,3452	12017,1714	63	114,04
45	0,00005	0,5	200	0,9637	0,9532	6147,7382	1938,9881	499	728,96
46	0,00005	0,75	200	0,9652	0,9486	6032,8318	2176,1142	499	726,7
47	0,00005	0,9	200	0,9095	0,9063	16197,7147	4310,2271	109	182,32
48	0,0002	0,5	500	0,7074	0,7043	41986,8591	10646,6421	48	223,75
49	0,0002	0,75	500	0,5598	0,5513	57068,0345	14499,8993	46	217,19
50	0,0002	0,9	500	0,5433	0,5324	49867,3846	13281,1319	499	2396,92
51	0,0001	0,5	500	0,9607	0,9504	6692,5389	2102,7813	328	1170,05
52	0,0001	0,75	500	0,8623	0,8558	22778,4898	5913,2075	73	305,29
53	0,0001	0,9	500	0,6762	0,6673	40937,1366	10507,3494	70	291,3
54	0,00009	0,5	500	0,9365	0,9301	10675,9911	2918,9981	193	719,06
55	0,00009	0,75	500	0,8885	0,8882	19614,2386	5071,6220	91	363,38
56	0,00009	0,9	500	0,8885	0,8767	19153,6635	5264,1784	150	568,4
57	0,00005	0,5	500	0,9639	0,9521	6057,7936	1936,1907	499	1708,28
58	0,00005	0,75	500	0,9323	0,9253	11548,6634	3162,4319	177	680,39
59	0,00005	0,9	500	0,8587	0,8531	22838,6127	6065,0620	150	567,52

Table 3.2: Results of batch learning.

The model with the best performance is the one with **ID=23**. Specifically, it outperforms the other models in terms of accuracy and loss, as it has a higher accuracy and a lower error.

3.3.1 Discussion

The following heatmaps, provide a visual representation of how accuracy and loss metric vary as the learning rate, the momentum and number of internal neurons change. These visualizations facilitate a better understanding of the trade-offs involved in selecting hyperparameters, helping to identify the optimal configurations for achieving the best performance.

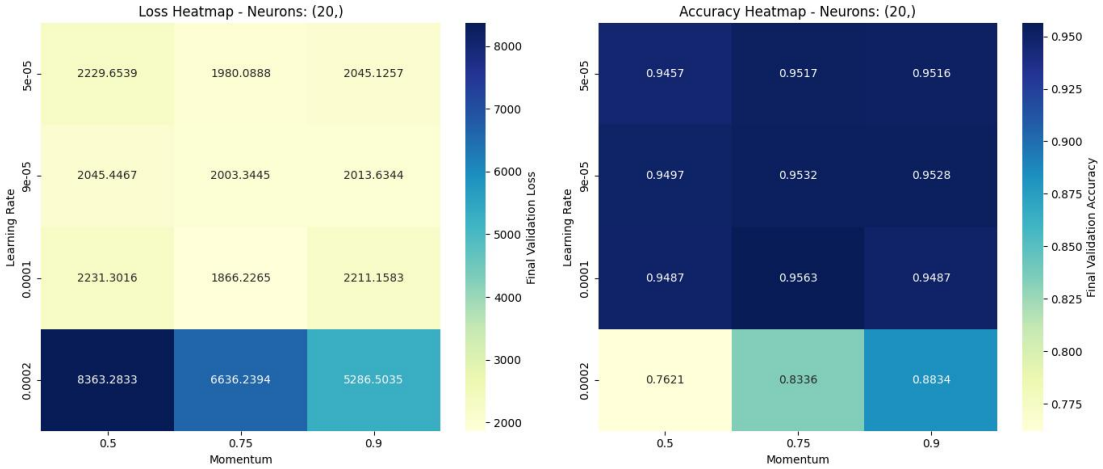


Figure 3.3: Heatmap of loss and accuracy of validation set with 20 neurons.

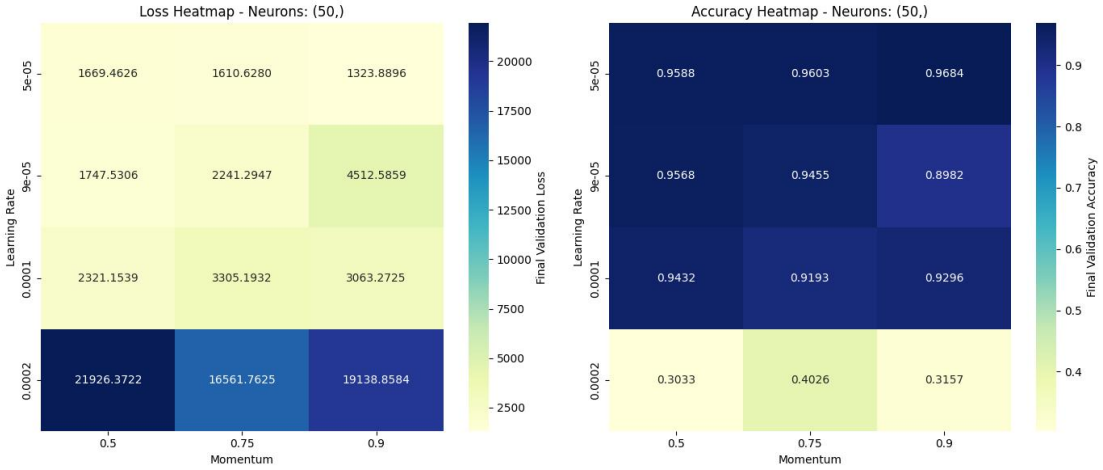


Figure 3.4: Heatmap of loss and accuracy of validation set with 50 neurons.

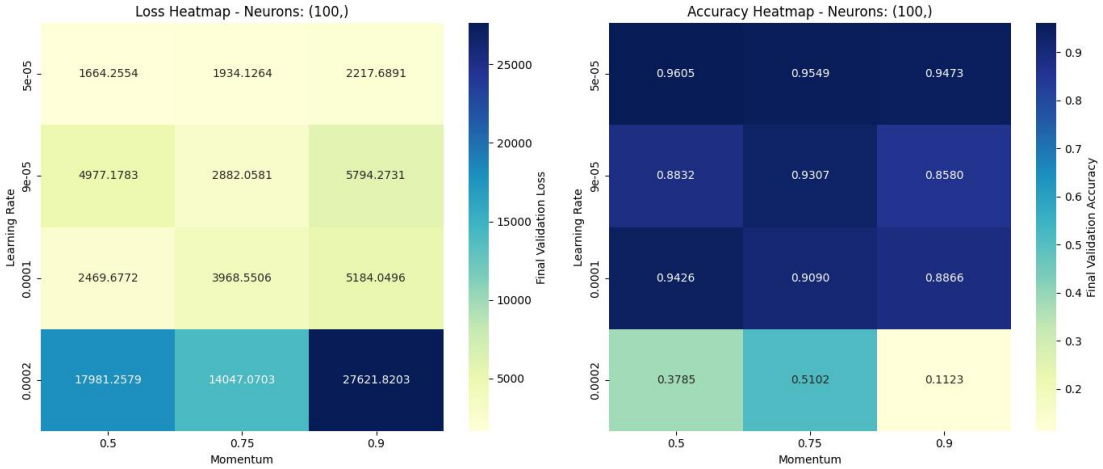


Figure 3.5: Heatmap of loss and accuracy of validation set with 100 neurons.

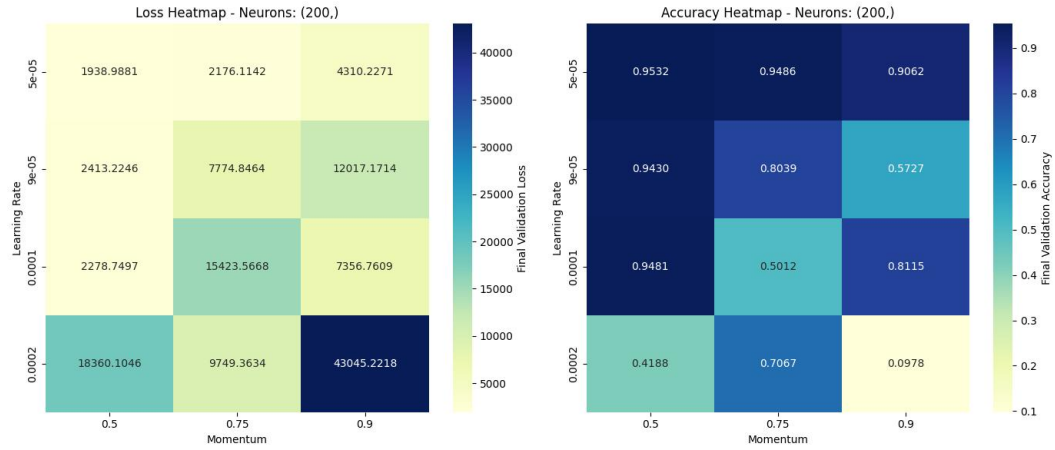


Figure 3.6: Heatmap of loss and accuracy of validation set with 200 neurons.

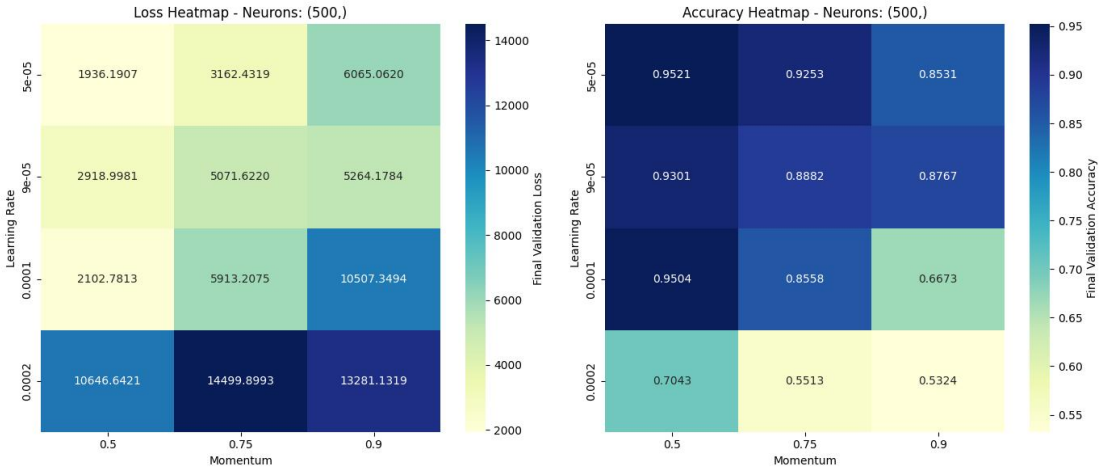


Figure 3.7: Heatmap of loss and accuracy of validation set with 500 neurons.

With 20 neurons (for accuracy $\mu = 0,9198$, $\sigma = 0,062$; for loss $\mu = 3242,6672$, $\sigma = 2224,44996$; for time $\mu = 100,89$, $\sigma = 53,7$; for epochs $\mu = 315,41$, $\sigma = 172,41$) excellent results are generally obtained in a very short time. However, excessively reducing the learning rate and momentum can slow convergence: in fact, training stops because the maximum number of epochs is reached. This implies that, with more epochs, the models could have been further refined. Therefore, to achieve good, if not better, results in a shorter time, it is sufficient to use a low learning rate combined with a higher momentum value.

When using 50 neurons (for accuracy $\mu = 0,7918$, $\sigma = 0,273$; for loss $\mu = 6618,5003$, $\sigma = 7728,1623$; for time $\mu = 160,12$, $\sigma = 97,94$; for epochs $\mu = 280,58$, $\sigma = 185,1$), the network's capacity increases, so a learning rate that is too high does not allow for good results. However, decreasing the learning rate improves performance. Furthermore, a higher momentum not only helps to refine the model but also allows training to finish in fewer epochs and with reduced execution time. Naturally, compared to the network with 20 neurons, the training time and number of required epochs increase, but there is an improvement in both accuracy and loss.

With 100 neurons (for accuracy $\mu = 0,7728$, $\sigma = 0,28$; for loss $\mu = 7561,8339$, $\sigma = 8114,1782$; for time $\mu = 195,98$, $\sigma = 155,71$; for epochs $\mu = 205$, $\sigma = 180,34$), it becomes clear that lower values are needed for both the learning rate and momentum. However, this results in an increase in execution time and the number of epochs required for convergence. With values not too low for η and too high for α the model converges to a good solution quickly. However, converging so quickly is counterproductive, as a more complex model with a larger search space isn't given enough time. A combination of a very low learning rate with a higher momentum provides good solutions in less time, but the results are not as good as in previous cases. Thus, using a smaller number of neurons could be preferable.

The same principle applies to networks with 200 (for accuracy $\mu = 0,7176$, $\sigma = 0,27$; for loss $\mu = 10570,3616$, $\sigma = 11596,24809$; for time $\mu = 238,31$, $\sigma = 236$; for epochs $\mu = 169,58$, $\sigma = 169,19$) and 500 neurons (for accuracy $\mu = 0,8072$, $\sigma = 0,153$; for loss $\mu = 6780,7912$, $\sigma = 4374,693171$; for time $\mu = 767,63$, $\sigma = 673,79$; for epochs $\mu = 193,67$, $\sigma = 162,93$). Even in these cases, low values for both the learning rate and momentum are necessary to obtain good results, but the training time increases significantly. Moreover, it can be seen how the results vary between 200 and 500 neurons when the learning rate and momentum are higher. This confirms what was said earlier: the model is too complex and is not given enough time to find a solution in a larger search space.

Therefore, from the results, it can be inferred that the problem is not too complex and that a relatively small number of neurons can provide better results in less time.

In summary:

- When the learning rate is too high (0.0002), performance worsens as the number of neurons increases. This could be due to the fact that the

network's capacity has increased (i.e., the model operates in a higher-dimensional space with a more complex error surface). A learning rate that is too high causes the model to "overshoot" the local minima, leading it to regions with meaningless error values. Additionally, in these cases, a higher momentum worsens the situation by increasing the speed at which weight updates are made, which are already too large (with the exception of networks with fewer neurons, where a higher momentum can improve performance).

- By reducing the learning rate, even slightly, better results are obtained (compared to the previous situation), even when using more neurons. In these circumstances, a moderate momentum is preferable to a higher one. Naturally, the problem persists because, although there is improvement compared to the previous case, performance continues to degrade.

- The same argument can be applied to smaller learning rates (e.g., $\eta = 0.00009$). That is, results improve, but the degradation issue persists as the number of neurons increases.

- Finally, with the smallest learning rate, significantly better results are obtained across all configurations. However, as the number of neurons increases, more time is naturally required for convergence. A higher momentum can help reduce execution time, but this comes at the cost of performance, especially with more than 100 neurons.

3.4 Refining the model

Grid search usually performs best when it is performed repeatedly. Let's consider, for example, the number of neurons. We found that the best value is 50. Therefore, we are underestimating the range in which the optimal number of neurons lies, and we should shift the grid and perform a new search with values relatively close to the one previously found. Therefore, a new grid search will be applied.

This time, the hyperparameters used are as follows:

- **Number of neurons**: $[45, 50, 55]$;
- **Learning rate**: $[0,000075, 0,00005, 0,00001]$
- **Momentum**: $[0.85, 0.9, 0.99]$

The results are shown in Table 3.4.

η	α	**Neurons**	A_{train}	A_{val}	$Loss_{train}$	$Loss_{val}$	**Epochs**	**Time**	$F1_{val}$
0,000075	0,85	45	0,9901	0,9616	2075,7620	1493,1516	413	244,71	0,9610
0,000075	0,9	45	0,9754	0,9477	4256,7740	2227,1435	419	305,12	0,9469
0,000075	0,99	45	0,8418	0,8412	31024,2442	7666,4982	18	27,53	0,8375

η	α	**Neurons**	A_{train}	A_{val}	$Loss_{train}$	$Loss_{val}$	**Epochs**	**Time**	$F1_{val}$
0,00005	0,85	45	0,9903	0,9687	2045,9989	1268,5334	499	374,58	0,9683
0,00005	0,9	45	0,9941	0,9669	1538,5000	1336,9097	485	265,57	0,9665
0,00005	0,99	45	0,9652	0,9378	5888,4497	3441,6600	305	173,28	0,9368
0,00001	0,85	45	0,9621	0,9538	6537,0710	1919,0955	499	263,59	0,9531
0,00001	0,9	45	0,9698	0,9581	5322,3816	1706,7441	499	261,1	0,9574
0,00001	0,99	45	0,9751	0,9449	4206,8785	2773,0672	499	263,6	0,9439
0,000075	0,85	50	0,9630	0,9393	6422,6541	2407,0976	297	179,2	0,9381
0,000075	0,9	50	0,9723	0,9453	4936,0554	2359,0292	357	213,02	0,9447
0,000075	0,99	50	0,9037	0,8950	18392,5993	5572,5250	158	117,49	0,8936
0,00005	0,85	50	0,9893	0,9645	2204,3267	1364,7825	499	300,96	0,9639
0,00005	0,9	50	0,9935	0,9650	1682,8802	1431,8487	419	245,98	0,9645
0,00005	0,99	50	0,9611	0,9356	6573,9487	3618,3203	259	153,85	0,9346
0,00001	0,85	50	0,9618	0,9515	6630,8066	1962,7934	499	281,31	0,9507
0,00001	0,9	50	0,9709	0,9586	5087,3264	1655,5040	499	280,06	0,9580
0,00001	0,99	50	0,9772	0,9449	3916,7225	2656,7496	499	286,14	0,9440
0,000075	0,85	55	0,9845	0,9569	3090,0872	1777,7045	447	263,9	0,9562
0,000075	0,9	55	0,9373	0,9188	10961,2400	3382,8096	196	118,65	0,9172
0,000075	0,99	55	0,7877	0,7872	41599,7788	10368,5659	26	24,74	0,7715
0,00005	0,85	55	0,9885	0,9648	2433,0146	1373,2498	499	278,25	0,9641
0,00005	0,9	55	0,9941	0,9673	1516,9470	1333,2397	471	270,87	0,9669
0,00005	0,99	55	0,9120	0,9048	20611,4905	5855,0922	49	37,55	0,9031
0,00001	0,85	55	0,9630	0,9553	6341,8511	1855,5056	499	280,3	0,9547
0,00001	0,9	55	0,9726	0,9617	4839,0507	1582,9477	499	277,65	0,9611
0,00001	0,99	55	0,9777	0,9462	3905,0975	2621,2787	499	278,03	0,9453

Table 3.4: Results of new grid search.

The row highlighted in blue represents the best configuration obtained from the previous grid search, while the row highlighted in green represents the model with the highest accuracy (on the validation set, of course). It should be noted that although the model highlighted in blue has slightly lower accuracy, it represents a good compromise between accuracy, loss, epochs, and execution time. However, since the chosen metric for selecting the best model is accuracy, we should opt for the green model. Additionally, since the difference in execution time is minimal, it is reasonable in this context to choose the model with higher accuracy (noting that the F1-score is also higher).

3.4.1 Testing

Now that we have obtained the best model from the tuning phase, we can proceed to test that model. First, we show in Figure 3.8 a plot of the accuracy and loss by epochs of the chosen model.

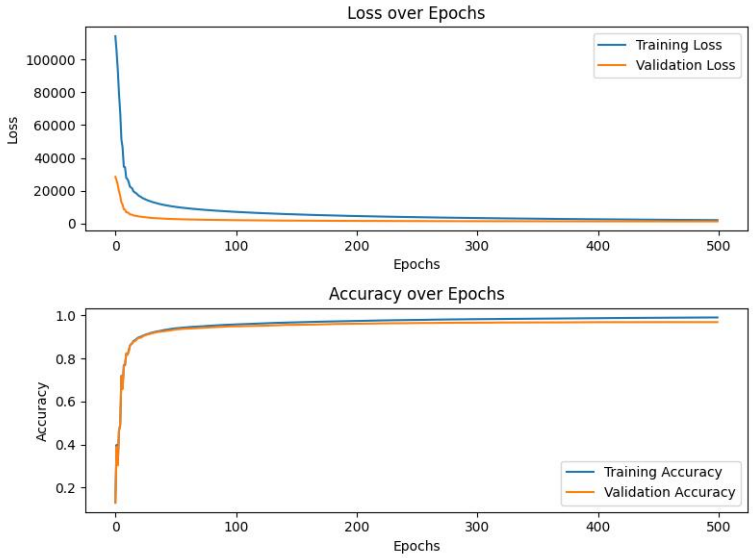


Figure 3.8: Plot of metrics of the selected model.

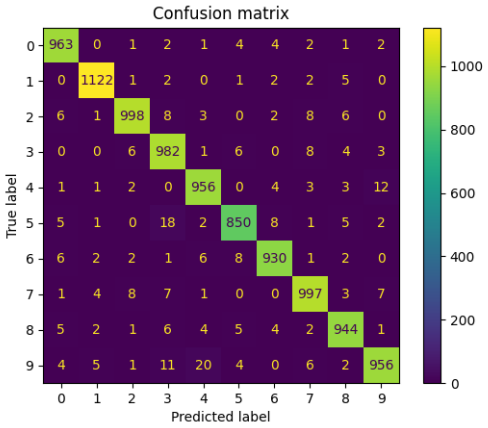


Figure 3.9: Confusion matrix using the test set.

Digit	P	R	F1	Support
Digit 0	0.97	0.98	0.98	980
Digit 1	0.99	0.99	0.99	1135
Digit 2	0.98	0.97	0.97	1032
Digit 3	0.95	0.97	0.96	1010
Digit 4	0.96	0.97	0.97	982
Digit 5	0.97	0.95	0.96	892
Digit 6	0.97	0.97	0.97	958
Digit 7	0.97	0.97	0.97	1028
Digit 8	0.97	0.97	0.97	974
Digit 9	0.97	0.95	0.96	1009
Accuracy			0.97	10000
Macro avg	0.97	0.97	0.97	10000
Weighted avg	0.97	0.97	0.97	10000

Table 3.6: Classification report of the best model.

As can be seen, there is 97% accuracy on the test. This result is in line with the accuracy obtained on training and validation. Therefore, the model was able to generalize to unseen data.

3.5 Learning mode comparison

In this section, the comparison of different learning modes is shown. Specifically, as the hyperparameters (number of neurons, learning rate and momentum)

change, the learning modes are evaluated.

So, we have for the *mini-batch* mode:

- **Number of neurons**: $[50, 100, 200]$
- **Learning rate**: $[0.002, 0.0001, 0.00009]$
- **Number of mini batches**: $[32, 64]$
- **Total combination**: 54.

For the *online* mode we have:

- **Number of neurons**: $[50, 100, 200]$
- **Learning rate**: $[0.2, 0.01, 0.009]$
- **Total combination**: 27

Therefore, we have $54 + 27 = 81$ combinations. Refer to the table in the previous section for batch mode combinations.

ID	**M**	**S**	**N**	η	α	A_{train}	A_{val}	L_{train}	L_{val}	$F1_{val}$	**E**	**T**
0	MB	32	0,002	0,5	50	0,9945	0,9688	1364,7610	1244,4643	0,9684	57	47,64
1	MB	32	0,002	0,75	50	0,9850	0,9620	2656,9510	1544,3294	0,9615	25	30,23
2	MB	32	0,002	0,9	50	0,9742	0,9568	4141,0213	1792,4596	0,9562	10	17,55
3	MB	32	0,0001	0,5	50	0,9850	0,9689	2902,6841	1279,8147	0,9686	499	361,25
4	MB	32	0,0001	0,75	50	0,9938	0,9708	1551,7590	1197,5848	0,9704	476	346,53
5	MB	32	0,0001	0,9	50	0,9939	0,9709	1550,5061	1234,6888	0,9706	186	144,22
6	MB	32	0,00009	0,5	50	0,9826	0,9664	3168,3188	1330,5394	0,9660	499	361,34
7	MB	32	0,00009	0,75	50	0,9941	0,9726	1489,3697	1170,4537	0,9722	499	325,15
8	MB	32	0,00009	0,9	50	0,9943	0,9700	1534,9457	1212,6117	0,9696	208	144,41
9	MB	64	0,002	0,5	50	0,9947	0,9696	1342,1757	1179,5713	0,9692	52	48,49
10	MB	64	0,002	0,75	50	0,9886	0,9677	2128,9547	1268,6859	0,9672	19	25,72
11	MB	64	0,002	0,9	50	0,9844	0,9663	2611,0152	1329,0614	0,9658	8	17,49
12	MB	64	0,0001	0,5	50	0,9846	0,9683	2933,6137	1318,3555	0,9679	499	365,12
13	MB	64	0,0001	0,75	50	0,9947	0,9702	1428,9398	1216,4268	0,9698	489	381,34
14	MB	64	0,0001	0,9	50	0,9950	0,9701	1378,7494	1194,9125	0,9697	204	162,01
15	MB	64	0,00009	0,5	50	0,9830	0,9668	3178,4910	1365,3478	0,9664	499	365,75
16	MB	64	0,00009	0,75	50	0,9938	0,9726	1533,0147	1134,5249	0,9722	499	367,71
17	MB	64	0,00009	0,9	50	0,9933	0,9690	1622,0438	1220,4752	0,9685	200	160,42
18	MB	32	0,002	0,5	100	0,9848	0,9600	2678,8667	1617,3687	0,9594	56	86,4
19	MB	32	0,002	0,75	100	0,9479	0,9327	8605,9908	3004,2319	0,9314	41	68,4
20	MB	32	0,002	0,9	100	0,9639	0,9419	5907,1564	2678,4522	0,9409	50	77,51
21	MB	32	0,0001	0,5	100	0,9886	0,9719	2293,0306	1115,8963	0,9715	499	606,03
22	MB	32	0,0001	0,75	100	0,9980	0,9768	899,0004	964,5565	0,9764	499	634,81
23	MB	32	0,0001	0,9	100	0,9985	0,9760	741,0071	988,9208	0,9757	227	320,02
24	MB	32	0,00009	0,5	100	0,9861	0,9715	2611,1361	1144,5713	0,9712	499	636,05
25	MB	32	0,00009	0,75	100	0,9969	0,9757	1047,0323	1023,6103	0,9754	499	680,01

ID	M	S	N	η	α	A_{train}	A_{val}	L_{train}	L_{val}	$F1_{val}$	E	T
26	MB	32	0,00009	0,9	100	0,9981	0,9754	834,9616	1003,2469	0,9750	232	326,45
27	MB	64	0,002	0,5	100	0,9982	0,9740	768,8280	1053,0041	0,9737	57	79,04
28	MB	64	0,002	0,75	100	0,9980	0,9720	697,9052	1170,7320	0,9716	39	62,75
29	MB	64	0,002	0,9	100	0,9817	0,9639	2950,0777	1506,3021	0,9634	8	31,81
30	MB	64	0,0001	0,5	100	0,9881	0,9729	2309,0402	1096,3926	0,9726	499	641,79
31	MB	64	0,0001	0,75	100	0,9975	0,9744	919,5121	1055,8791	0,9741	499	711,84
32	MB	64	0,0001	0,9	100	0,9993	0,9764	620,2269	964,2275	0,9762	248	285,59
33	MB	64	0,00009	0,5	100	0,9858	0,9715	2651,6600	1157,5667	0,9712	499	530,46
34	MB	64	0,00009	0,75	100	0,9972	0,9747	1054,7422	1021,4680	0,9743	499	526,19
35	MB	64	0,00009	0,9	100	0,9989	0,9751	668,5325	969,9942	0,9747	264	295,19
36	MB	32	0,002	0,5	200	0,9769	0,9559	3951,2206	1864,0809	0,9553	51	130,06
37	MB	32	0,002	0,75	200	0,9673	0,9466	5254,5705	2256,6948	0,9458	33	95,01
38	MB	32	0,002	0,9	200	0,9493	0,9344	7819,5873	2972,1358	0,9334	31	88,85
39	MB	32	0,0001	0,5	200	0,9897	0,9733	2083,4755	1057,9360	0,9730	499	956,05
40	MB	32	0,0001	0,75	200	0,9986	0,9792	708,3864	871,6554	0,9789	499	1007,75
41	MB	32	0,0001	0,9	200	0,9991	0,9786	600,5323	900,7189	0,9783	225	500,12
42	MB	32	0,00009	0,5	200	0,9880	0,9744	2318,3610	1076,0916	0,9741	499	983,78
43	MB	32	0,00009	0,75	200	0,9981	0,9773	845,5338	930,9430	0,9770	499	957,88
44	MB	32	0,00009	0,9	200	0,9991	0,9763	631,0764	931,9507	0,9760	237	487,33
45	MB	64	0,002	0,5	200	0,9954	0,9731	1210,3206	1105,1223	0,9727	51	140,8
46	MB	64	0,002	0,75	200	0,9810	0,9591	3122,0559	1657,3423	0,9584	25	87,23
47	MB	64	0,002	0,9	200	0,9619	0,9419	5944,1383	2371,4601	0,9411	16	68,84
48	MB	64	0,0001	0,5	200	0,9899	0,9753	2069,8438	1036,8811	0,9750	499	1051,73
49	MB	64	0,0001	0,75	200	0,9985	0,9774	725,0054	924,3524	0,9771	499	1068,08
50	MB	64	0,0001	0,9	200	0,9997	0,9781	452,2088	899,5928	0,9778	260	607,72
51	MB	64	0,00009	0,5	200	0,9880	0,9728	2337,9104	1079,4785	0,9725	499	1189,38
52	MB	64	0,00009	0,75	200	0,9980	0,9768	862,3216	925,5782	0,9764	499	1143,29
53	MB	64	0,00009	0,9	200	0,9994	0,9798	501,2297	876,9490	0,9795	267	607,22
54	O		0,5	0,5	50	0,9098	0,9050	20427,9385	5401,9563	0,9049	38	2246,21
55	O		0,5	0,75	50	0,7648	0,7624	54325,4214	13726,1278	0,7307	8	1233,44
56	O		0,5	0,9	50	0,1408	0,1396	325450,2666	81635,6729	0,0725	13	1532,61
57	O		0,01	0,5	50	0,9960	0,9723	1013,6394	1158,5839	0,9721	15	1112,88
58	O		0,01	0,75	50	0,9942	0,9702	1124,8243	1241,1559	0,9699	11	1188,71
59	O		0,01	0,9	50	0,9820	0,9649	2704,5836	1492,5982	0,9647	6	915,44
60	O		0,005	0,5	50	0,9935	0,9705	1451,2254	1159,6445	0,9703	21	1325,72
61	O		0,005	0,75	50	0,9959	0,9705	1009,6827	1200,7953	0,9703	16	1414,51
62	O		0,005	0,9	50	0,9875	0,9674	1993,9374	1351,4217	0,9671	7	963,66
63	O		0,5	0,5	100	0,8590	0,8564	103877,3273	26449,2588	0,8514	40	4721,65
64	O		0,5	0,75	100	0,7684	0,7668	201392,5698	50194,5737	0,7585	29	3930,7
65	O		0,5	0,9	100	0,2027	0,2056	552429,9282	136574,7866	0,1126	2	1417,23
66	O		0,01	0,5	100	0,9996	0,9771	356,3404	910,0030	0,9769	19	1828,72
67	O		0,01	0,75	100	0,9995	0,9783	288,8608	974,8339	0,9781	12	1857,21
68	O		0,01	0,9	100	0,9998	0,9784	98,1204	1105,6341	0,9782	13	1818,89
69	O		0,005	0,5	100	0,9990	0,9773	616,3789	892,9081	0,9770	26	2184,16
70	O		0,005	0,75	100	0,9996	0,9780	377,7269	900,5452	0,9778	18	2526,87

ID	M	S	N	η	α	A_{train}	A_{val}	L_{train}	L_{val}	$F1_{val}$	E	T
71	O		0,005	0,9	100	0,9997	0,9771	209,6162	967,0191	0,9769	12	1736,33
72	O		0,5	0,5	200	0,8326	0,8342	161984,4812	39730,7934	0,8327	25	9104,57
73	O		0,5	0,75	200	0,6508	0,6580	359922,2232	88026,3175	0,6377	12	7160,76
74	O		0,5	0,9	200	0,2074	0,2165	828577,9726	204634,0785	0,1254	0	3747,37
75	O		0,01	0,5	200	0,9996	0,9802	299,0250	808,0045	0,9800	17	6552,24
76	O		0,01	0,75	200	0,9999	0,9796	160,2749	824,8652	0,9794	12	6249,45
77	O		0,01	0,9	200	0,9999	0,9812	60,3101	968,8172	0,9810	11	5914,49
78	O		0,005	0,5	200	0,9997	0,9786	306,9921	853,7927	0,9784	33	10174,29
79	O		0,005	0,75	200	0,9997	0,9792	298,7972	839,6273	0,9790	17	6850,16
80	O		0,005	0,9	200	0,9996	0,9801	227,6556	807,3245	0,9799	9	4971,65

Table 3.8: M denotes the modality of learning, S denotes the size of the mini-batches, N denotes the number of neurons, T the time, E the epochs, B stands for Batch, MB stands for Mini-batch and O for Online.

As can be seen, the results are on average better than in batch mode. In particular, with the mini-batch mode, regardless of the complexity of the network, learning rate and time, excellent results are obtained. With the online mode, however, the results are more variable when the learning rate is too high. However, by decreasing it, excellent results can also be obtained with this mode. However, as evidenced by the purple lines, there are cases where the error on the validation set is greater than the error on the training set. Interestingly, learning has not stopped. This is because the validation error continued to decrease but not at the same rate as the training error. Such a scenario may have occurred because with these two learning modes, the weights are updated for each mini-batch/sample in the training set, leading the model to fit better on the data on the has just been updated. Indeed, as can be seen in the following sections, the model responded well on unseen data by achieving accuracy in line with that of training/validation.

3.5.1 Testing best model obtained from mini-batch learning

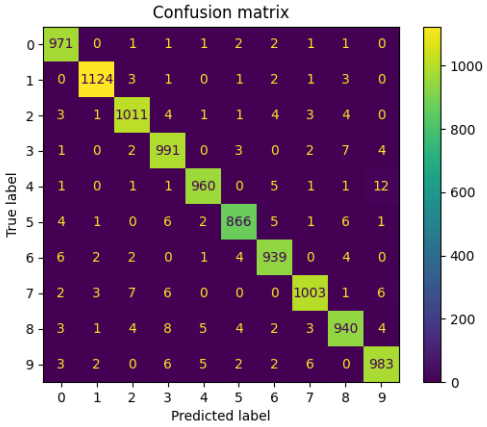


Figure 3.10: Confusion matrix using the test set (on model 53).

Digit	P	R	F1	Support
Digit 0	0.98	0.99	0.98	980
Digit 1	0.99	0.99	0.99	1135
Digit 2	0.98	0.98	0.98	1032
Digit 3	0.97	0.98	0.97	1010
Digit 4	0.98	0.98	0.98	982
Digit 5	0.98	0.97	0.98	892
Digit 6	0.98	0.98	0.98	958
Digit 7	0.98	0.98	0.98	1028
Digit 8	0.97	0.97	0.97	974
Digit 9	0.97	0.97	0.97	1009
Accuracy			0.98	10000
Macro avg	0.98	0.98	0.98	10000
Weighted avg	0.98	0.98	0.98	10000

Table 3.10: Classification report of the best model (53).

3.5.2 Testing best model obtained from online learning

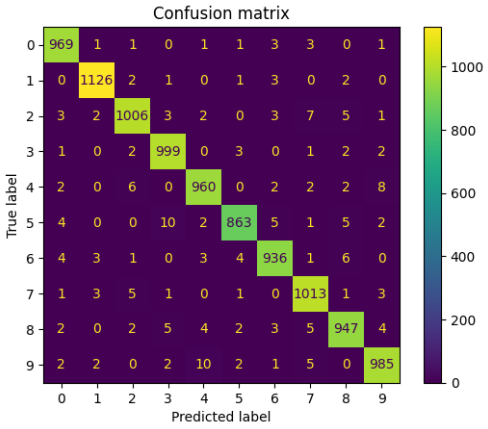


Figure 3.11: Confusion matrix using the test set (on model 83).

Digit	P	R	F1	Support
Digit 0	0.98	0.99	0.98	980
Digit 1	0.99	0.99	0.99	1135
Digit 2	0.98	0.97	0.98	1032
Digit 3	0.98	0.99	0.98	1010
Digit 4	0.98	0.98	0.98	982
Digit 5	0.98	0.97	0.98	892
Digit 6	0.98	0.98	0.98	958
Digit 7	0.98	0.99	0.98	1028
Digit 8	0.98	0.97	0.97	974
Digit 9	0.98	0.98	0.98	1009
Accuracy			0.98	10000
Macro avg	0.98	0.98	0.98	10000
Weighted avg	0.98	0.98	0.98	10000

Table 3.12: Classification report of the best model (83).

3.5.3 Discussion

Finally, we show the differences between the various modalities in terms of number of neurons, time taken, epochs used, accuracy and error (on validation set).

In terms of **accuracy**, it is evident that the worst-performing mode is full-batch, which shows a higher percentage of models with accuracy below the median. In contrast, mini-batch modes with both 32 and 64 batches have more concentrated values around the median, which is over 96 percent. Furthermore, the 64 mini-batch mode maintains fairly consistent performance as the number of neurons increases, while the 32 mini-batch mode shows greater variability as the architecture becomes more complex. The online mode, on the other hand, produces less uniform results, worsening as the number of neurons increases. However, it is notable that the median accuracy (in the later cases) is higher than in the 32/64 mini-batch modes, suggesting that 50% of the models have higher accuracy than in the mini-batch modes. Nonetheless, this difference remains quite subtle.

The same observation can be applied to the **loss**.

Regarding the number of **epochs** required, the online mode, as expected, takes a much smaller number of epochs to complete the learning. In other cases, however, the situation changes depending on the architecture. When the network has 50 neurons in the hidden layer, the batch mode turns out to require the most epochs. This could be due to the fact that the network is not very complex and, in addition, the mini-batch mode by updating the weights more frequently allows convergence to be achieved faster. However, as the number of neurons increases, the batch mode seems to be the fastest (compared to the mini-batch mode). This result is due to the fact that the network is more complex and the number of parameters increases. As a result, frequently updating the weights may require more epochs.

Finally, in terms of execution **time**, it can be seen that the online mode is the slowest. Although relatively few epochs were sufficient with that mode, the time taken to finish the training is too high. The other modes, in this regard, are more efficient. In particular, the batch mode turns out to be the fastest. The 32/64 mini-batch modes as the complexity of the network increases require more execution time.

The mini-batch mode proves to be a winning strategy for several reasons. First, it shows an accuracy distribution more concentrated around very high values (above 96%), with low variability in the results. This means that mini-batch not only produces models that perform well but does so with greater consistency. Additionally, the number of epochs and the time required to achieve such performances are relatively short compared to the batch mode, which, although the fastest in terms of time per epoch, does not guarantee models with the same level of accuracy. Therefore, mini-batch mode truly represents the best **trade-off** for achieving high-performance models in reasonably short times.

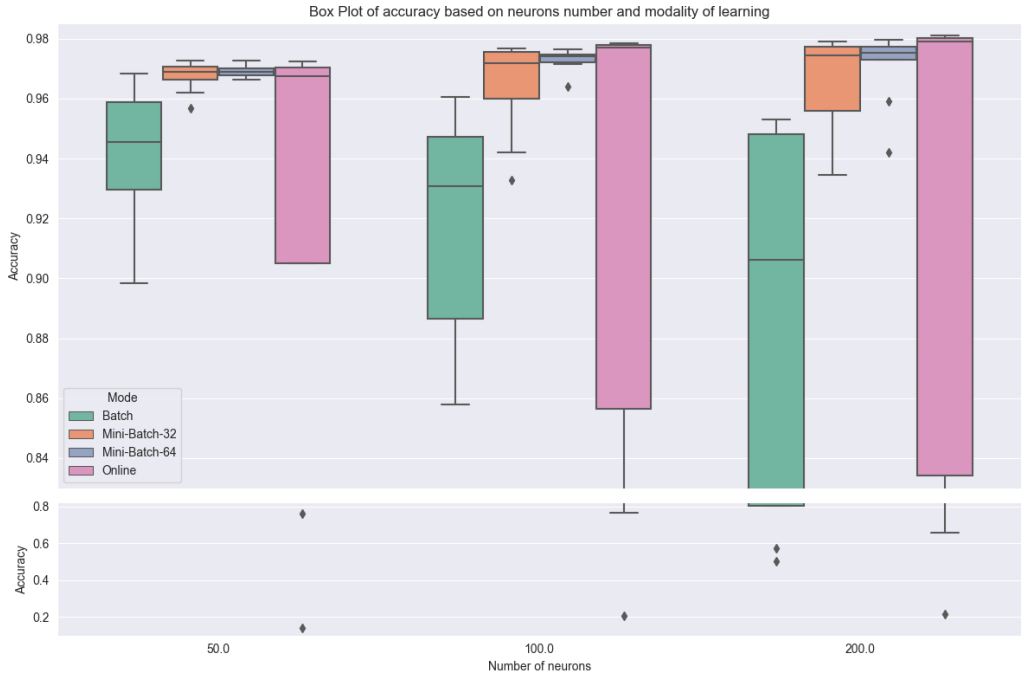


Figure 3.12: Box plot of accuracies

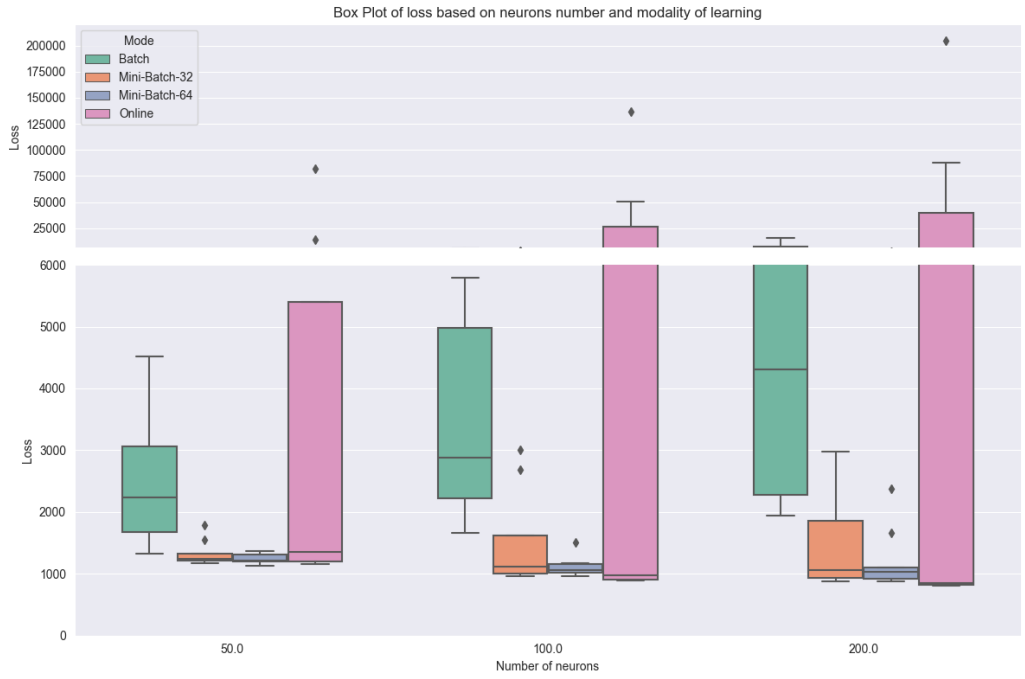


Figure 3.13: Box plot of errors

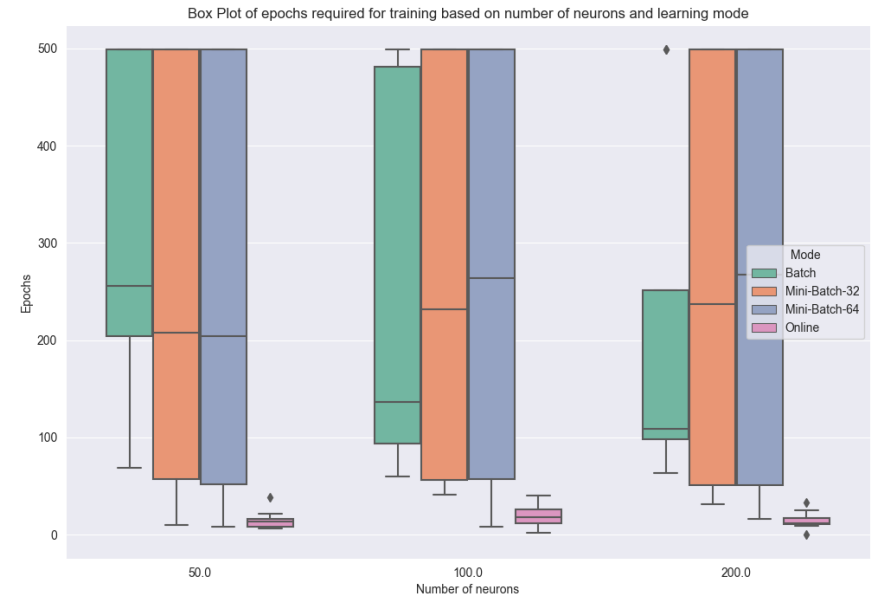


Figure 3.14: Box plot of epochs

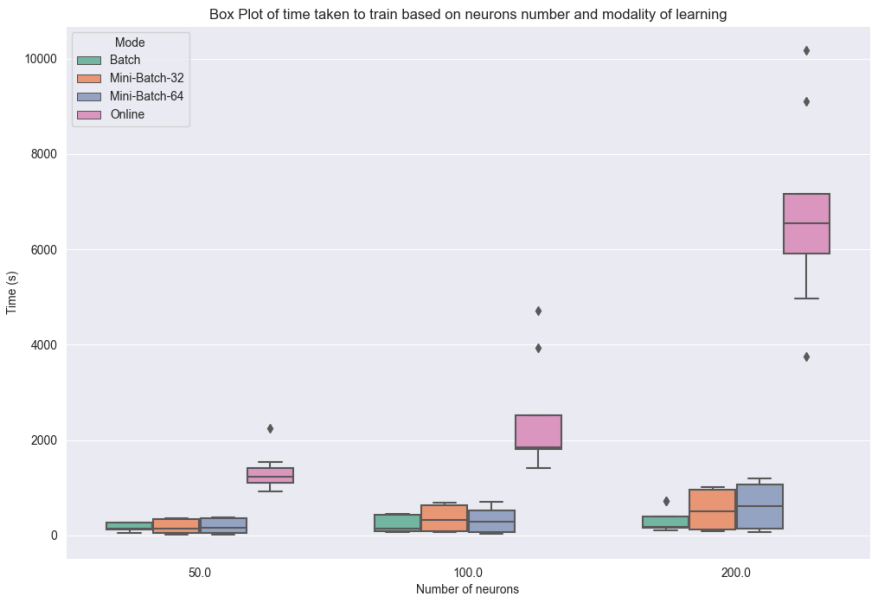


Figure 3.15: Box plot of time

3.6 Vanilla gradient descent vs Gradient descent with momentum

It might be interesting to show the difference in learning when using vanilla gradient descent and gradient descent with momentum.

First of all, the following aspect should be considered: in gradient descent with momentum, for each value of the learning rate there were three different values of momentum. However, in the classic gradient descent, this does not occur, as it is as if the momentum is 0. Therefore, for each value of the learning rate, three different runs were performed, in order to make the comparison between the two methods more fair (i.e., the number of rows is the same). The same hyperparameters shown in the table with batch learning (3.2) were considered, except for the group with 500 neurons, in order to not overload the plots.

The results of training with classic gradient descent are shown in Table 3.14. These results will be summarized in several box plot to enhance clarity (starting from Fig. 3.26).

η	**Run**	**Neurons**	A_{train}	A_{val}	L_{train}	L_{val}	**Epochs**	**Time**
0,0002	1	20	0,8566	0,8575	24182,57428	6222,620606	85	27,48
0,0002	2	20	0,9136	0,9074	14557,73224	4027,229082	109	38,66
0,0002	3	20	0,8993	0,8916	17657,79416	4757,070726	116	39,24
0,0001	1	20	0,9104	0,9062	15082,89598	4028,737552	83	29,69
0,0001	2	20	0,9180	0,9113	13803,33969	3734,707839	89	31,62
0,0001	3	20	0,9180	0,9106	13889,91564	3776,624118	96	34,17
0,00009	1	20	0,9215	0,9143	13128,87418	3587,862931	126	43,19
0,00009	2	20	0,9355	0,9255	10622,46359	3048,449212	207	65,91
0,00009	3	20	0,9310	0,9228	11579,92311	3251,91542	165	54,31
0,00005	1	20	0,9453	0,9328	9211,36151	2773,048965	499	149,46
0,00005	2	20	0,9467	0,9361	9076,090261	2691,946856	499	153,93
0,00005	3	20	0,9474	0,9378	8888,493047	2669,9572	499	162,56
0,0002	1	50	0,7579	0,7559	33322,7153	8463,9280	87	60,8
0,0002	2	50	0,7998	0,8003	28063,3070	7225,8571	59	48,22
0,0002	3	50	0,8671	0,8613	22179,9981	5826,9311	110	79,19
0,0001	1	50	0,9655	0,9503	5877,9716	2004,4177	499	301,42
0,0001	2	50	0,9426	0,9338	9419,1777	2755,0928	227	138,09
0,0001	3	50	0,9457	0,9344	8958,2055	2674,2864	219	123,3
0,00009	1	50	0,9633	0,9506	6225,7812	2044,5289	499	262,17
0,00009	2	50	0,9691	0,9568	5398,1949	1810,4838	499	259,71
0,00009	3	50	0,9681	0,9547	5530,9546	1853,0701	499	263,09
0,00005	1	50	0,9560	0,9464	7457,9588	2259,7878	499	1697,13
0,00005	2	50	0,9552	0,9431	7658,3026	2318,0112	499	265,25
0,00005	3	50	0,9553	0,9443	7552,7855	2297,1978	499	289,1
0,0002	1	100	0,9262	0,9131	13110,4168	3799,3103	172	157,83
0,0002	2	100	0,7525	0,7507	33143,2514	8477,0185	116	110,21

η	**Run**	**Neurons**	A_{train}	A_{val}	L_{train}	L_{val}	**Epochs**	**Time**
0,0002	3	100	0,8564	0,8533	23485,9049	6142,8702	73	75,6
0,0001	1	100	0,9638	0,9498	6069,3967	2051,2412	499	421,57
0,0001	2	100	0,9687	0,9561	5428,2155	1865,0810	499	423,64
0,0001	3	100	0,9252	0,9184	12749,8555	3464,5103	133	126,59
0,00009	1	100	0,9701	0,9585	5105,5559	1721,4431	499	425,8
0,00009	2	100	0,9685	0,9558	5306,1974	1797,0445	499	423,06
0,00009	3	100	0,9292	0,9230	11932,3427	3346,2271	131	134,55
0,00005	1	100	0,9582	0,9474	7066,9632	2180,5151	499	424,78
0,00005	2	100	0,9575	0,9448	7264,1598	2225,5444	499	428,2
0,00005	3	100	0,9574	0,9468	7152,5006	2197,7442	499	424,56
0,0002	1	200	0,9294	0,9183	11688,4295	3376,2959	172	267,85
0,0002	2	200	0,9109	0,9019	14700,0559	4111,5460	127	204,49
0,0002	3	200	0,8647	0,8618	22604,7736	5923,0258	96	159,56
0,0001	1	200	0,9260	0,9201	13040,7557	3606,5493	135	215,45
0,0001	2	200	0,9679	0,9535	5530,4493	1910,3491	499	711,72
0,0001	3	200	0,9646	0,9509	6034,7466	2020,1048	499	713,37
0,00009	1	200	0,9639	0,9513	6200,5436	2026,4282	499	712,2
0,00009	2	200	0,9431	0,9331	9397,6829	2777,0842	250	377,68
0,00009	3	200	0,9622	0,9495	6324,5715	2079,0490	499	709,92
0,00005	1	200	0,9570	0,9457	7364,9608	2227,4947	499	711,3
0,00005	2	200	0,9590	0,9461	6964,1113	2188,3668	499	709,73
0,00005	3	200	0,9575	0,9459	7277,9468	2201,5645	499	705,74

Table 3.14: Training results using classic gradient descent.

3.6.1 Testing

After grid search we choose the best model, that is, the one that obtained the highest accuracy on the validation set. This model is the one highlighted in green.

The results obtained in testing are shown below.

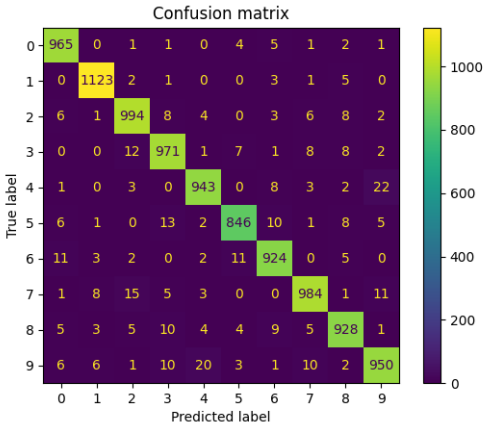


Figure 3.16: Confusion matrix using the test set.

Digit	P	R	F1	Support
Digit 0	0.96	0.98	0.97	980
Digit 1	0.98	0.99	0.99	1135
Digit 2	0.96	0.96	0.96	1032
Digit 3	0.95	0.96	0.96	1010
Digit 4	0.96	0.96	0.96	982
Digit 5	0.97	0.95	0.96	892
Digit 6	0.96	0.96	0.96	958
Digit 7	0.97	0.96	0.96	1028
Digit 8	0.96	0.95	0.96	974
Digit 9	0.96	0.94	0.95	1009
Accuracy			0.96	10000
Macro avg	0.96	0.96	0.96	10000
Weighted avg	0.96	0.96	0.96	10000

Table 3.16: Classification report of the best model (vanilla gradient descent).

Thus, the model trained without momentum achieved 96% accuracy on the test-set, while the model with momentum 97% (see Section 3.4.1). However, the time taken by the model that made use of momentum was 374.58. Therefore, not only the model is more accurate, but it achieved a better result in a shorter time (about 13%).

3.6.2 Discussion

First, some plots (Fig. 3.19) are shown below to highlight how accuracy and loss vary when the momentum is present or not.

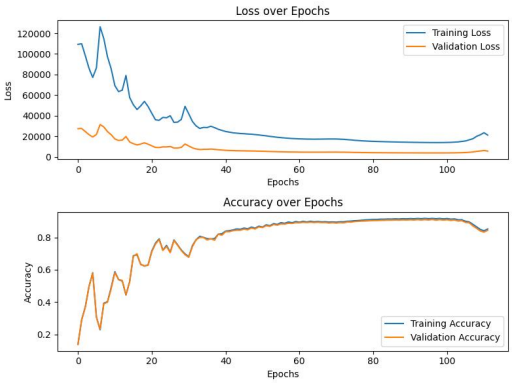


Figure 3.17: Model with $\eta = 0.0001$, $run = 3$

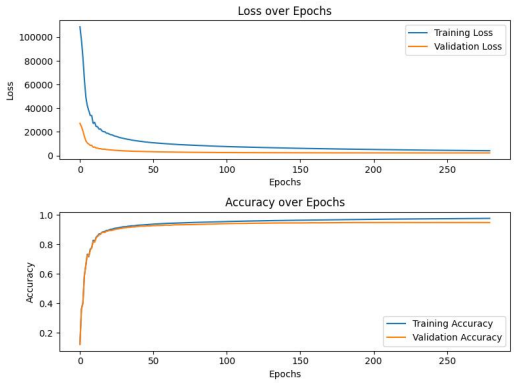


Figure 3.18: Model with $\eta = 0.0001$, $\alpha = 0.9$

Figure 3.19: Comparison of model metrics with and without momentum when the network has 20 internal neurons.

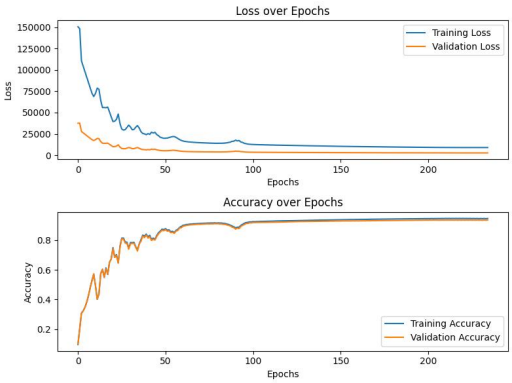


Figure 3.20: Model with $\eta = 0.0001$, $run = 3$

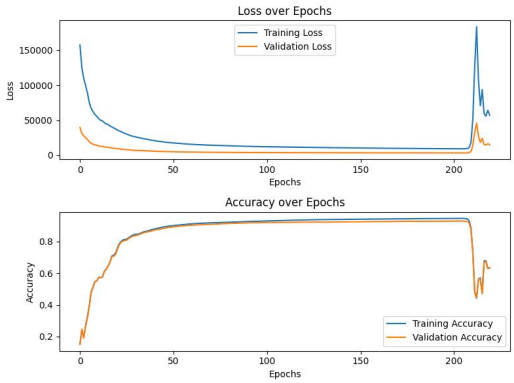


Figure 3.21: Model with $\eta = 0.0001$, $\alpha = 0.9$

Figure 3.22: Comparison of model metrics with and without momentum when the network has 50 internal neurons.

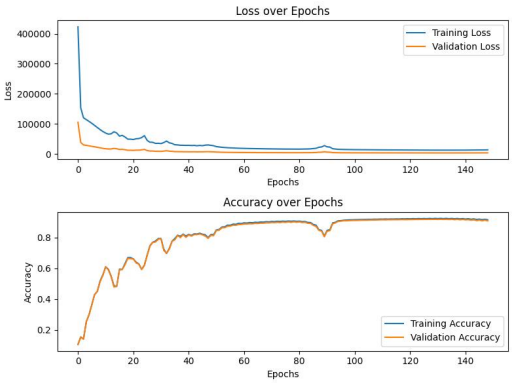


Figure 3.23: Model with $\eta = 0.0001$, $run = 3$

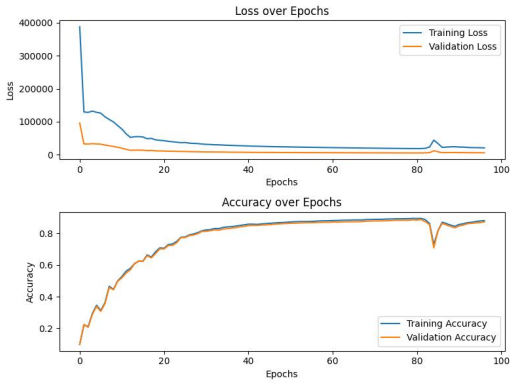


Figure 3.24: Model with $\eta = 0.0001$, $\alpha = 0.9$

Figure 3.25: Comparison of model metrics with and without momentum when the network has 100 internal neurons.

As you can see from the plots, the momentum helped reduce the oscillations, making the model more stable and allowing faster convergence.

Finally we show the difference in terms of accuracy, loss, epochs and time spent on training.

From Box Plot 3.26, it is clear that the performance of models that did use momentum as an update rule is more variable, with a higher number of outliers. This occurs because combining momentum with a learning rate that is too high does not allow the model enough time to stabilize (i.e., the early stopping condition is triggered too soon). Nevertheless, when the architecture is simpler (20 hidden neurons), the overall performance is better.

Obviously, the variability in performance is also due to the fact that momentum changes along with the learning rate, whereas in the vanilla gradient descent, it is as if momentum is always set to 0. For this reason, models without momentum appear more stable, but this is just a didactic example.

The same reasoning applies to the loss.

In terms of epochs and execution time, momentum tends to extend the training time only when the architecture is simpler. However, this could be due to the fact that momentum has helped to make the models more stable, as evidenced by the higher accuracy. As the number of hidden neurons increases, it becomes clear that momentum allows for faster convergence.

In conclusion, when the learning rate is high, classical gradient descent is shown to be more effective because it gives the network time to converge. In contrast, a lower learning rate, combined with momentum, helps achieve better results in a shorter time.

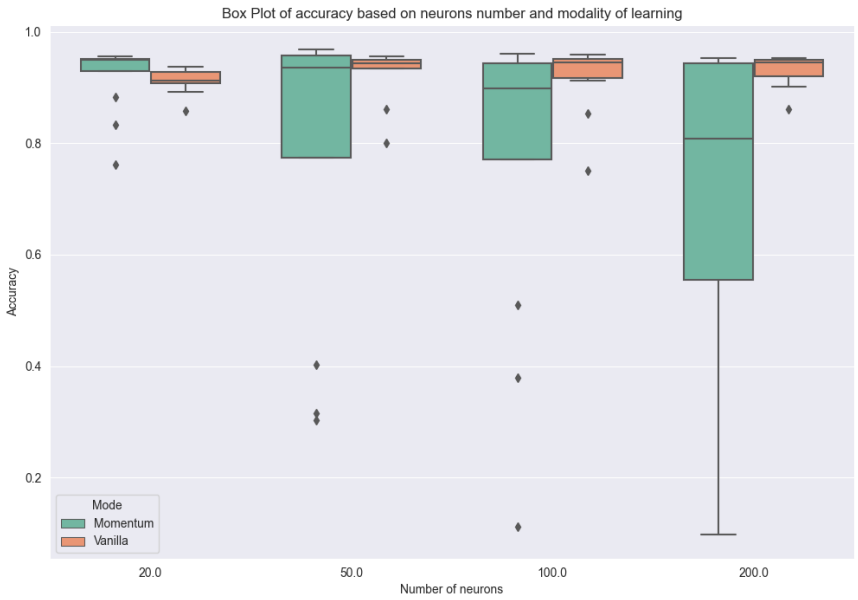


Figure 3.26: Box plot of accuracies of vanilla gradient descent and gradient descent with momentum.

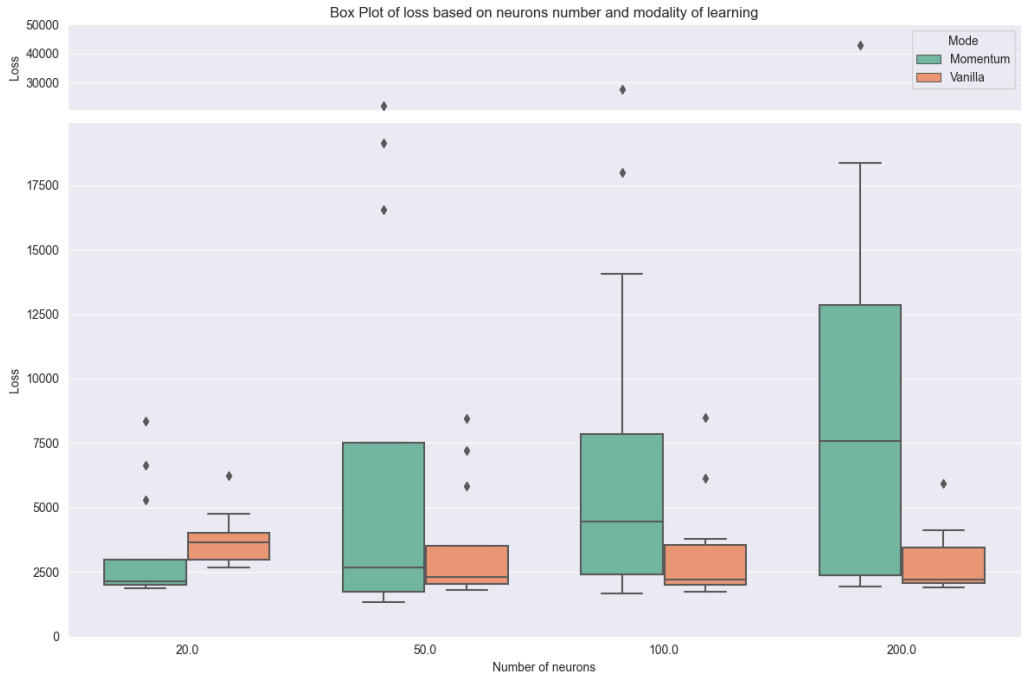


Figure 3.27: Box plot of errors of vanilla gradient descent and gradient descent with momentum.

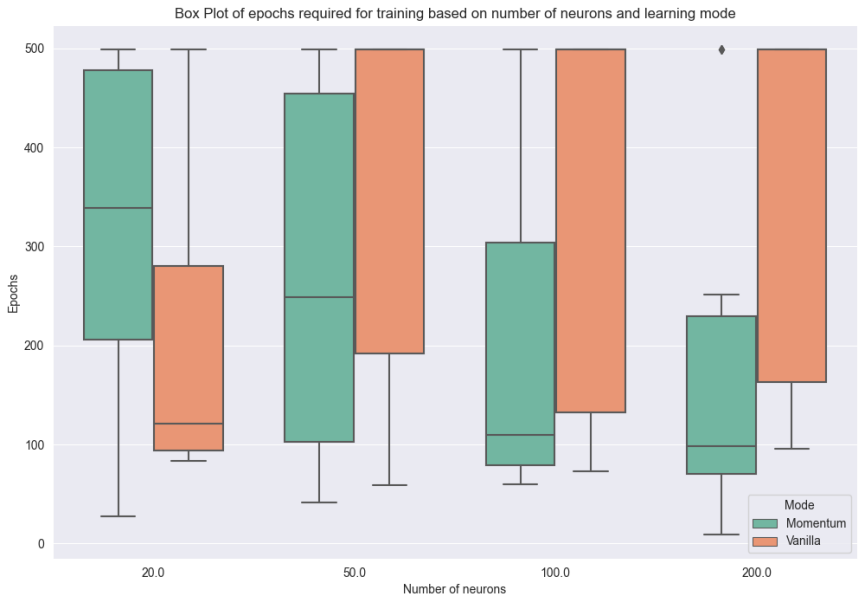


Figure 3.28: Box plot of epochs of vanilla gradient descent and gradient descent with momentum.

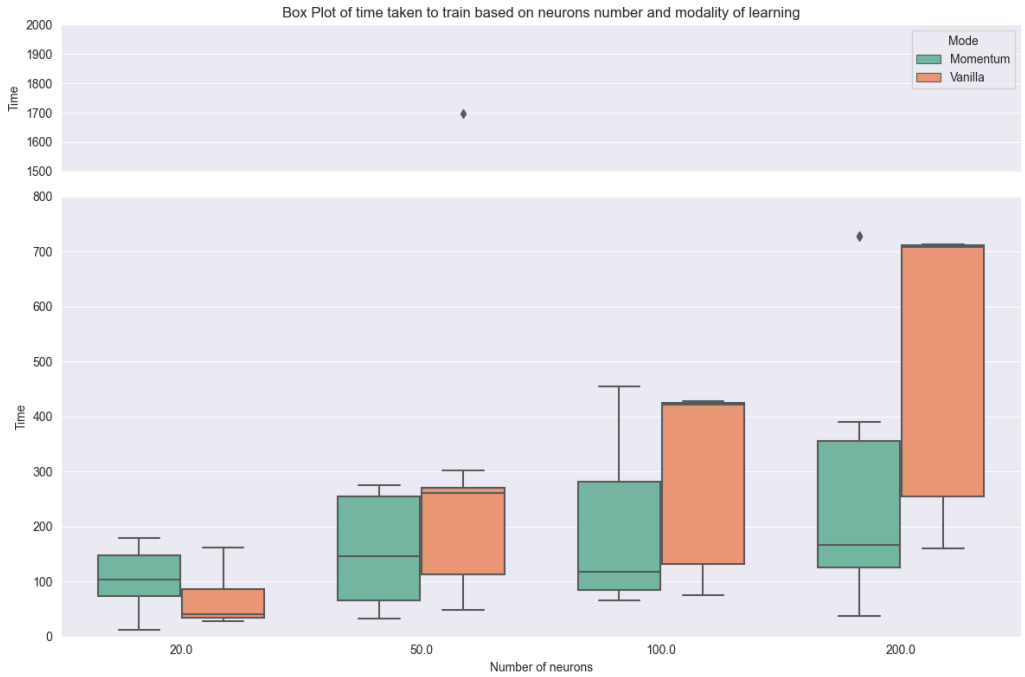


Figure 3.29: Box plot of time of vanilla gradient descent and gradient descent with momentum.