Development of a feed forward full connected Neural Network

D. De Vita, S. Falangone, M. Moraca 5/2020

1 Abstract

This work describes theoretically and practically a general Feed Forward Full Connected Neural Network. We aim to explain the logic behind Artificial Neural Networks (ANN) and provide a developed and extensible code on activation and error functions, number of layers and output functions. Partial derivatives' calculation is achieved through commonly known algorithms of forward and backward propagation. The implemented update policy is the Resilient Back Propagation algorithm. We analyze the early stopping criteria techniques from [1] to avoid the overfitting issue. In the last sections we provide some test cases analyzing the Average accuracy and F-measure linked to Generalization Loss (GL) and the Progress Quotient (PQ) stopping criteria. The code is appended to this work to allow as much reproducibility as possible.

2 Introduction

The developed Feed Forward Full Connected Neural Network has been applied to the recognition of handwritten digits images; this can easily be handled as a multi-class classification problem, where the classes are the ten digits from *0* to *9*.

$$C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Each class is symbolized by a neuron in the last layer, and its output represents the probability that given an input, it is classified as the corresponding class. Obviously we focus on classification, instead of regression, because the output expected from the Neural Network is the label associated to the highest probability of the corresponding class; on the other hand, regression returns a continuous value that is an estimation of some quantity. Note that the developed Neural Network can achieve regression changing the activation functions for each layer and the error function that shall be minimized.

Modularity was achieved through Object-Oriented design. Implementations for activation and error functions, early stopping criteria were provided in corresponding objects and can be further extended to reader's needs.

A limit of this work is the nature of full connected and feed forward structure; a user cannot choose how much connected is a layer to the next.

To feed a neural network a dataset of labeled inputs shall be provided: a collection of samples of what the network should try to recognize, each labelled with the correct class to which it belongs; in our case the dataset is a collection of handwritten digits images (28x28): The MNIST

database (**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology database). The dataset is partitioned in two disjoint subsets:

- **Training Set (60.000 images):** of which the network will be fed, more than once eventually;
- **Test Set (10.000 images):** on which the network will be evaluated at the end of the training. Through this set of images we compute the evaluation metrics.

Out of the Training Set we extract another partition denominated **Validation Set**. This last shall be used to test the network behaviour during the training; the main idea is to verify, at the end of each epoch, how the network would behave.

3 Feed Forward Neural Networks

To explain precisely what is a Feed Forward Neural Network, we need to specify that generally a Neural Network is an object which is characterized by a finite number of inputs and neurons, a finite number of connections (typically one-way connections) which link two nodes *j* and *i*.
To every arc (i, j), starting in node *i* and ending in node *j*, is associated a weight, formalized as w_{ji}.
Every node computes an output that becomes the input of the linked nodes. To calculate the outputs, in addition to input values and edges' weights, nodes have a bias value and an activation function *g*. The output value calculated by a node is produced from the application of the activation function to the sum of the dot product of weights with inputs and the bias. This values is called *activation*:

$$a_i = \left(\sum_j w_{ij} x_j \right) + b_i$$

where *j* runs on each connection from *j-th* node to the node *i*. Therefore the output value of *i-th* node is:

$$z_i = g(a_i)$$

The order by which is possible to calculate the output values is called *activation sequence* and it can be synchronous, or asynchronous. A Feed Forward Network is an example of a Neural Network which has two principal properties:

- There aren't any cycles. It's possible to check the satisfaction of this property by assigning, in ascending topological order, natural numbers to inputs and to the neurons and verify that every node can take connections only from elements which have a lower number;
- The activation sequence of each neuron is asynchronous and it follows the topological sorting enforced by connections.

There is a subclass of Feed Forward Neural Network that organizes neurons in disjoint subsets (named *layers*) l_1, l_2,..., l_r called *Feed Forward Multi Layer Neural Network*. Every node

belonging to a layer, we say l_k, can receive connections only from neurons of the immediately previous layer l_{k-1} and can provide outputs only to neurons belonging to the successive layer l_{k+1}. Through this organization we can call internal nodes all those who belong to the layers l_1, l_2,..., l_{r-1} and output nodes all those who belong to the layer l_r.
Another subclass is the *Feed Forward Full Connected Neural Network*, which is the one we implemented for this project, where every neuron of a certain layer n_i^{l-1} is connected to all the neurons belonging to the next layer n_i^l.

3.1 Activation Functions

As previously specified, the Activation Function *g* is part of each neuron and it defines the output of that node given an input, the activation value *a*.
Each node of every layer could use a different Activation Function, however this is rarely done; usually, the same activation function is used for each node of the same layer, or for all the hidden layers and differs only on the output layer.
Although there are no mandatory constraints on the behaviour of an activation function, there are some guidelines based on what kind of result is expected.
The range of the activation function is often required to be $(0, 1)$, extremely useful in a probabilistic interpretation, or $(-1, 1)$.
In order to approximate with a single hidden layer as much as possible the behaviour of the unknown Model function, activation functions should be non-polynomial. Moreover, when the output of a neuron establishes how much the input activates it, another often desired property is the monotonic behaviour of the activation function.
In our context, it was decided to implement a small variety of Activation Functions which will be explained briefly below.

Identity Function

$$g(x) = x$$

The Identity Function is probably the most simple and naive activation function; it returns the same input it gets. The range of this function is the whole $\mathbb{R}$, it is monotone, continuous and the main advantage provided is the short computation time.

Heavside Function** a.k.a **Binary Step Function

$$g(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

The Heavside Function is another simple and efficient option; the range of this function is $\{0, 1\}$, it is a mere simulation of the historical Perceptrons behaviour.

Sigmoid Function** a.k.a **Logistic Function

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

The Sigmoid Function is one of the most notorious among the simple activation functions; it behaves as continuous and monotone variation of the Heavside Function, granting $(0, 1)$ as range. In order to keep the python implementation numerically stable, when the exponential function could cause an overflow or underflow, we used "tend-to-0" and "tend-to-1" output value.

Softmax Function

$$g_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \text{for } j = 1, ..., J$$

The Softmax Function differs from the previous functions because it does not work on a single node but on a vector (the whole layer). The main characteristic of this function is that the sum of all outputs obtained on a vector will always be 1.
This feature suites perfectly our purpose when used as the Output Layer function, interpreting the output of every node i (10 in our output layer) as the probability that the given input is an instance of the class C_i.
Unfortunately, many attempts to stabilize numerically this function, such as scaling or shifting the exponential argument, lead to a serious loss of performance.

3.2 Error Functions

Error functions E (or $E(\underline{\theta})$), also known as *cost functions* or *loss functions*, are used to quantify how distant we are from the desired behaviour or, to be more specific, how poorly we are simulating the Model function.
These are commonly binary functions that consider both the output obtained (the prediction) and the output expected (the target or label); It is reasonable to consider this as the function to minimize; in order to do this, to select only Error Functions that offer $\mathbb{R}^+$ as range where a value of 0 means that no error was committed on the predictions.
As for the activation ones, there are no formal requirements about Error Functions but it would be preferable to use functions that can be written as an average. This would be useful because the back propagation algorithm allows to consider only one input at a time when it calculates the derivatives; and so the Error Function could be interpreted as:

$$E = \frac{1}{N} \sum_{n=1}^{N} E^{(n)} \qquad \text{where } E^{(n)} \text{ is the Error on the } n^{th} \text{ input}$$

As mentioned before, Error functions are binary functions and accept as arguments: the prediction $\underline{y}$, and the expectation $\underline{t}$
In our work we decided to consider only two different Error functions:

Sum of Squares Function

$$E^{(n)}(\underline{y}, \underline{t}) = \frac{1}{2} \sum_{k=1}^{|C|} (y_k^n - t_k^n)^2$$

The *Sum of Squares Function* represents precisely what the Error function should do; it computes the squared euclidean distance between the given output and the correct classification.

This quantity is then divided by 2 in order to simplify the derivative computation.
It is a commonly used function in regression problems (where t is continuous) and being, as the name suggests, a sum of squares always returns non-negative values.

Cross-entropy Function

$$E^{(n)}(\underline{y}, \underline{t}) = -\sum_{k=1}^{|C|} t_k^n \log y_k^n$$

The *Cross-entropy Function* is mostly used in classification problems (where t is discrete). This function too returns only non-negative values because $y_k^n, t_k^n \in [0, 1]$ for all k and n; so $\log y_k^n$ is always non-positive obtaining a sum of all non-positive values; the minus before the sum grants non-negative values.
In order to ease the computational effort of the back-propagation, it was decided to implement a "special version" for both Softmax and Cross-Entropy Functions. These versions are only to be used together.

3.3 Forward and Back Propagation

Remembering that our goal is to minimize the error function $E(\underline{\theta}) = E(\underline{w}, \underline{b})$ (in the following, the biases are treated as weights), the gradient vector $\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..)$ shall be calculated for each parameter (weights and biases). Global error can be decomposed as the sum of N local errors (one for each input)

$$E(\underline{\theta}) = \sum_{n=1}^{N} E^{(n)} \tag{1}$$

For the linearity of derivative operator, generic change of error function in w_{ij} can be calculated as:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{n=1}^{N} \frac{\partial E^{(n)}}{\partial w_{ij}}. \tag{2}$$

Understood the minimization problem, forward and back propagation algorithms can be used to achieve partial derivatives of the error function in linear time on the number of weights[1] .

For each node *i* of the Neural Network we define $\delta_i = \frac{\partial E^{(n)}}{\partial a_i}$, from this quantity it can be proved that

$$\frac{\partial E}{\partial w_{ij}} = \delta_i z_j. \tag{3}$$

If *k* is a node of the output layer

$$\delta_k = f'(a_k) \frac{\partial E^{(n)}}{\partial y_k}. \tag{4}$$

Otherwise for a generic node i, defining (i, j) as the arc from neuron j to i

$$\delta_i = (\sum_k w_{ki} \delta_k) * g'(a_i). \tag{5}$$

[1] A naive implementation could compute derivatives perturbing each weight by Δw_{ij} and calculate $\frac{\partial E}{\partial w_{ij}} = \frac{E(\underline{w} + vect(\Delta w_{ij})) - E(\underline{w})}{\Delta w_{ij}}$ where vect defines a null vector of the same dimension of w that has component ij equal to Δw_{ij}. Unfortunately time complexity is quadratic on number of weights.

Defined these formulas, a simple algorithm to compute partial derivates of weights and biases on input $(x^{(n)}, t^{(n)})$ is:

1. Forward propagate on input $\underline{x}$ calculating activations and output for the Neural Network;
2. Using (4) calculate δ_i for the last layer;
3. Using (5) calculate δ_{i-1} going backward to the first layer;
4. Using (3) compute partial derivate as a simple product for each parameter.

4 Update Weights policies

When we say *"train a feed forward neural network"* we mean to choose the right value for each outgoing arc from a neuron, which is located inside a layer, to neurons of successive layer, with goal of minimizing the error function on the training set. To achieve this, we need to follow an iterative process based on training loops, called *epochs*, which updates the weights values and is executed until a stopping condition is reached.
To understand if the stopping condition is reached, which indicates that a good degree of generalization of a neural network has been gained, we have to evaluate the error function on the training set and validation set.

We should consequently wonder how to encode all the training process, where should be located the update weights as well as the evaluation of the error on training set and validation set.
In this section we show three algorithms which answer to the above questions and we focus on the main differences between them. Each different algorithm presents a solution that influences the amount of weight derivatives accumulated over different images of the training set: Batch and Minibatch approaches sum the derivatives of each weight among different images before updating the weights; on the other hand, the Online approach updates right after each and every image of the training set. This choice is based on a tradeoff between computational effort and performance.
In the end, we introduce the update rule we follow to update the weights (Rprop).

4.1 Batch

In this approach the update of weights and biases is done after the calculation of weights' derivatives of error function on the whole training set.

Algorithm 1 Batch learning

1: **procedure** BATCH
2: $epochs \leftarrow 0$
3: **while** not stopping criteria **do**
4: $\frac{\partial E}{\partial w_{ij}}^{t} \leftarrow 0$
5: $epochs \leftarrow epochs + 1$
6: **for** $n = 1 \rightarrow$ `training set size` **do**
7: `forwardpropagation()`
8: `backpropagation()`
9: **for all** w_{ij} **do**
10: $\frac{\partial E}{\partial w_{ij}}^{n} \leftarrow \delta_i * z_j$
11: $\frac{\partial E}{\partial w_{ij}}^{t} \leftarrow \frac{\partial E}{\partial w_{ij}}^{t} + \frac{\partial E}{\partial w_{ij}}^{n}$
12: **end for**
13: **end for**
14: `updating weights`
15: `Error evaluation on training set`
16: `Error evaluation on validation set` ▷ Useful for stopping criteria
17: **end while**
18: **end procedure**

A *con* of this algorithm is that, to work properly, the whole dataset has to be loaded in memory, this could be a problem for very large datasets.

4.2 Online

To avoid the issue of the previous strategy, a possible solution is to use the online learning algorithm. This approach updates the weights after each observation in the training set causing to fully upgrade the whole network on every input. Since the weights are updated every time an input is observed, this algorithm gives a good generalization to the neural network at the cost of a higher computational complexity.

Algorithm 2 Online learning

```
1: procedure ONLINE
2:     epochs ← 0
3:     while not stopping criteria do
4:         epochs ← epochs + 1
5:         for n = 1 → training set size do
6:             forwardpropagation()
7:             backpropagation()
8:             for all w_ij do
9:                 (∂E/∂w_ij)^n ← δ_i * z_j
10:            end for
11:            updating weights
12:        end for
13:        Error evaluation on training set
14:        Error evaluation on validation set        ▷ Useful for stopping criteria
15:    end while
16: end procedure
```

4.3 Minibatch

The minibatch algorithm is a tradeoff between the approaches seen above, it tries to take the advantages of the online learning method avoiding an elevate computational complexity by updating the weights after chunk of the dataset, loaded in memory like the batch algorithm, with a fixed size. It's important to note that a new hyperparameter is introduced in this approach: the dimension of minibatches.

Algorithm 3 Minibach learning

procedure MINIBACH
  $epochs \leftarrow 0$
  $miniBatchSize \leftarrow \frac{\texttt{training set size}}{miniBatchNumber}$
  while not stopping criteria **do**
    $\frac{\partial E}{\partial w_{ij}}^{t} \leftarrow 0$
    $miniBatchIterations \leftarrow 0$
    $epochs \leftarrow epochs + 1$
    for $n = 1 \rightarrow$ `training set size` **do**
      `forwardpropagation()`
      `backpropagation()`
      for all w_{ij} **do**
        $\frac{\partial E}{\partial w_{ij}}^{n} \leftarrow \delta_i * z_j$
        $\frac{\partial E}{\partial w_{ij}}^{t} \leftarrow \frac{\partial E}{\partial w_{ij}}^{t} + \frac{\partial E}{\partial w_{ij}}^{n}$
      end for
      $miniBatchIterations \leftarrow miniBatchIterations + 1$
      if $miniBatchIterations \geqslant miniBatchSize$ **then**
        `updating weights`
        $miniBatchIterations \leftarrow 0$
        $\frac{\partial E}{\partial w_{ij}}^{t} \leftarrow 0$
      end if
      end for
      `Error evaluation on training set`
      `Error evaluation validation set` ▷ Useful for stopping criteria
    end while
  end procedure

4.4 Resilient Back Propagation

Based on the chain rule, the backpropagation algorithm calculates the derivative and then modifies the corresponding weight as

$$w_{ij}^{t+1} = w_{ij}^{t} - \eta \frac{\partial E}{\partial w_{ij}}^{t} \tag{6}$$

This first approach defines a hyperparameter η known as *learning rate*, its use is to scale the derivative in updating the corresponding weight. A common problem associated is that in this approach η is fixed and determines the speed (and so the time) to achieve convergence in finding the minimum of the Error function. Choosing a small value of η allows a major precision but the price to pay is a computation time that could be unacceptable. On the other side, a value too high of *learning rate* might bring oscillations around a local minimum, something that should be avoided.

In literature, a first solution to this problem was proposed with Gradient Descent variant algo-

rithm; summing a momentum term Δw_{ij}^t to the previous weight.

$$\Delta w_{ij}^t = -\eta \frac{\partial E}{\partial w_{ij}}^t + \mu \Delta w_{ij}^{t-1} \tag{7}$$

Where a second hyperparameter, $\mu \in (0, 1)$, is introduced, namely the *momentum coefficient*. This heuristic is considered to avoid Gradient Descent's problems, unfortunately literature has shown some cases where this characteristic does not hold and μ is problem dependent just like the learning rate.

Resilient Back Propagation comes from the need to separate a direct influence of the derivative values on the update one considering some Δ whose value is defined just by the sign of derivatives.

In their work, Riedmiller and Braun introduce for each weight the corresponding **update-value** Δ_{ij} which determines the changing w_{ij} during the learning. Being an adaptive schema, the update-value evolves on local error function.

$$\Delta_{ij}^t = \begin{cases} \eta^+ \Delta_{ij}^{t-1}, & \text{if } \frac{\partial E}{\partial w_{ij}}^t * \frac{\partial E}{\partial w_{ij}}^{t-1} > 0, \\ \eta^- \Delta_{ij}^{t-1}, & \text{if } \frac{\partial E}{\partial w_{ij}}^t * \frac{\partial E}{\partial w_{ij}}^{t-1} < 0, \\ \Delta_{ij}^{t-1}, & \text{else} \end{cases} \tag{8}$$

Setting $0 < \eta^- < 1 < \eta^+$, the explanation of this formula is that if the derivative suddenly changes sign, then the minimum is missed and so we wish to reduce oscillation because last update was too large; if the sign remains the same then it's possible to speedup to fasten convergence. At the beginning we impose $\Delta_{ij}^0 = 0.1$. Established the update-value, we define the **update-weight**

$$\Delta w_{ij}^t = \begin{cases} -\Delta_{ij}^t, & \text{if } \frac{\partial E}{\partial w_{ij}}^t > 0 \\ +\Delta_{ij}^t, & \text{if } \frac{\partial E}{\partial w_{ij}}^t < 0 \\ 0, & \text{else} \end{cases} \tag{9}$$

The update is then performed as follows:

$$w_{ij}^t = w_{ij}^{t-1} + \Delta w_{ij}^t \tag{10}$$

If there is a change in sign of the derivative, namely $\frac{\partial E}{\partial w_{ij}}^t * \frac{\partial E}{\partial w_{ij}}^{t-1} < 0$, then the previous update-weight is restored with the previous one:

$$\Delta w_{ij}^t = -\Delta w_{ij}^{t-1} \tag{11}$$

After backtracking, we impose $\frac{\partial E}{\partial w_{ij}}^{t-1} = 0$ to avoid a second derivative change sign in next iteration.

5 Early Stopping

Understood the principle behind the training, and therefore the learning, of a neural network it could seem intuitive to keep feeding it, in order to achieve better performance. Even if the

dataset is scarce it is always possible to reiterate over many *epochs*; it could seem that the only reason to limit the number of epochs is the computation time.
However, feeding a neural network always with the same samples will quickly lead to a condition known as *over-fitting*; this means that the network is learning too much from those samples and it's lacking in generalization.
To avoid this issue, in this essay we have studied three *early-stopping criteria*; two of which have been implemented and tested. In general, all of the strategies to avoid over-fitting use a partition of the given training set only to check how the network behaves when it encounters an unknown input. This strategy allows the network to acknowledge its ability to generalize and to stop before it starts to recognize only input images.
Such partition of the dataset is composed of:

- **Training Set.** of which the network is fed;
- **Test Set.** on which the network is evaluated at the end of the training;
- **Validation Set.** on which the network is tested during the training.

The last one, in particular, is the main actor to handle this issue: a small set of data which the network is not trained on, but tested after each epoch. The error obtained on the *Validation Set*, or ***Validation Error (*** E_{val}***)***, is then used to trace over epochs; our goal is to find its global minimum.
Of course, being the number of epochs potentially infinite, it is impossible to find a global minimum of the error on validation set. It's adequate then to a look for a local minimum, iterating until the Validation Error starts growing instead of reducing, formally until the epoch e such as:

$$E_{val}(e) > E_{val}(e-1)$$

Of course, this naive strategy does not offer a valuable solution to the problem; mostly because it does not consider a eventual better local minimum located beyond an uphill. A possible solution to this has been proposed by [1]

In his work, Prechelt studies this issue and proposes three *stopping criteria* using three parameters:

- ***Generalization Loss*** **(**GL_α**)**
- ***Progress Quotient*** **(**PQ_α**)**
- ***Consecutive Increase (***UP_s***)***

In our work only the GL_α and PQ_α *criteria* have been implemented, but all three of them have been studied in order to gain and offer a more complete observation of the problem.
Among all the metrics that are used in these *criteria* it is important to introduce as soon as possible $E_{opt}(t)$; as formerly specified these work relies on the Validation Error E_{val} in order to

monitor whether or not the network is losing generality.
The $E_{opt}(t)$ value is defined as the lowest Validation Error encountered in the first t epochs.

$$E_{opt}(t) = \min_{t' \leq t} E_{val}(t')$$

Generalization Loss

$$GL_\alpha : \text{stop on epoch } t \text{ such as: } GL(t) > \alpha$$

$$\text{where } GL(t) = 100 \cdot \left(\frac{E_{val}(t)}{E_{opt}(t)} - 1 \right)$$

As it is evinced, $GL(t)$ is the proportion of the current Validation Error over the best known. Being by definition $E_{opt}(t) \leq E_{val}(t)$ it always holds that $GL(t) \geq 0$, returning 0 when the current error is a new minimum. Experimentally, we observed that this strategy suggests to stop in a few epochs even with a relatively high α like 20 (considering that the author of the paper used 5 as highest α).
The author himself suggested this approach when the desiderata is: "To maximize the probability of finding a 'good' solution".

Progress Quotient

$$PQ_\alpha : \text{stop on epoch } t \text{ such as: } \frac{GL(t)}{P_k(t)} > \alpha$$

$$\text{where } P_k(t) = 1000 \cdot \left(\frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \cdot \min_{t'=t-k+1}^{t} E_{tr}(t')} - 1 \right)$$

This *criteria* is a bit more complicate; it accounts a strip of k consecutive epochs and evaluates not only the *Validation Error*, but the *Training Error* as well. Author's interpretation of the $P_k(t)$ value is "how much was the average training error during the strip larger than the minimum training error during the strip?".
Opposed to GL approach, Prechelt recommends the PQ strategy if is required "To maximize the average quality of solutions" and "if the network overfits only very little"

During our work, we gave two different interpretations of the PQ strategy. The former, to which we refer as ***PQ_disjoint***, considers $\frac{n}{k}$ number of strips, where n is the training set size and k the length of the strips; these are adjacent and disjoint, so the PQ value is only considered on those iterations t such as:
$\frac{t}{k} = p \in \mathbb{N}^+$ (e.g the exact divsion).
The second interpretation, named ***PQ_overlap***, considers for each iteration $t \geq k$ the strip composed of the last k elements.
Of course, PQ_disjoint behaves exactly like PQ_overlap on those iterations where the former is defined; however, being evaluated more times, PQ_overlap has a higher chance to reach the *stop point* earlier; on the other hand PQ_disjoint has shown a greater reluctance to stop, reaching the end of the number of maximum epochs allowed before suggesting to stop.

Consecutive Increase

$$UP_s : \text{stop on epoch } t \text{ such as:}$$

This *criteria* does not consider the entity of the uphill encountered after a local minimum, but only the amount of "time" since the last time the Validation Error has lowered.

6 Assessments and Tests

Because of covering all the possible parameters in a test would have required an exponential number of test cases, some parameters were fixed while, one at time, was modified.

6.1 Evaluation Metrics

In order to read and understand the results we obtained, it was decided to compute a variety of evaluation metrics. To accomplish this task, it was decided to calculate all of the most notorious evaluation metrics and to pack them in a bean class.
To fully understand the meaning of the evaluation metrics it's necessary to introduce the concepts of *True/False Positives/Negatives*; these are usually meant to be evaluated on a single class classification, while in a multiclass instance it's necessary to compute them for each class $C_i \in C$.

- **True Positives (TP)**: Tests that have been classified as belonging to a class, and that actually belong;
- **True Negatives (TN)**: Tests that have been classified as **non-belonging** to a class, and that actually **do not** belong;
- **False Positives (FP)**: Tests that have been classified as belonging to a class, and that actually **do not** belong; (Mistake)
- **False Negatives (FN)**: Tests that have been classified as **non-belonging** to a class, and that actually belong. (Miss)

To our purposes we decided to keep note of the **Average Accuracy** and **Micro F-score**; however, in order to understand their meaning and application it is important to know where these come from. For starters, the ***Accuracy***:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

is the ratio of the correct tests on the total amount. In a multi-class classification problem, this value is computed on each class, and is usually extremely high due to the True Negatives that constitute the most of the outcomes.
The same issue goes with the ***Average Accuracy***, which gives a unique value to summarize the Accuracy of the Neural Network.

$$Accuracy_{AVG} = \sum_{i=1}^{|C|} \frac{TP_i + TN_i}{TP_i + TN_i + FP_i + FN_i}$$

Other two standard measures in classification are ***Precision*** and ***Recall***:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

These are peculiar because alone aren't useful enough. ***Precision*** can tell us how many of the items identified with a certain class actually belong to that class; while ***Recall*** tells us how many of the items that actually belong to a class are identified as such. Commonly both are combined in a single value named ***F-score***:

$$F\text{-}score = \frac{(\beta^2 + 1) \cdot Precision \cdot Recall}{\beta^2 \cdot Precision + Recall}$$

Also known as F-measure or F_1 score, this reaches the optimum value ($F = 1$) when Precision and Recall are maximized ($P = 1$ and $R = 1$). All the three of these are standard metrics for binary classification problems so, just like the ***Accuracy***, are computed for each class. To have a unique value that includes every class there are two strategies: ***Macro Measures:*** *Macro Precision, Macro Recall and Macro F-score*: defined as the average of these on all the classes. ***Micro Measures:*** *Micro Precision, Micro Recall and Micro F-score*: defined as the global metrics as follows.

$$Precision_\mu = \frac{\sum_{i=1}^{|C|} TP_i}{\sum_{i=1}^{|C|} TP_i + FP_i}$$

$$Recall_\mu = \frac{\sum_{i=1}^{|C|} TP_i}{\sum_{i=1}^{|C|} TP_i + FN_i}$$

$$F\text{-}score_\mu = \frac{(\beta^2 + 1) \cdot Precision_\mu \cdot Recall_\mu}{\beta^2 \cdot Precision_\mu + Recall_\mu}$$

6.2 Tests

Hyperparameters used for tests below:

- for the R-prop $\eta^+ = 1.2, \eta^- = 0.5$
- batch size 1 (Online approach)
- threshold inside GL $\alpha = 1.0$
- strip size and threshold for PQ disjoint and PQ overlap $stripSize = 5, \alpha = 0.5$

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.976	0.882	10
784x80x10	0.975	0.873	5
784x100x10	0.978	0.892	6
784x120x10	0.979	0.894	8
784x140x10	0.976	0.88	2
784x250x10	0.967	0.876	3

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.976	0.882	14
784x80x10	0.974	0.869	14
784x100x10	0.978	0.892	14
784x120x10	0.979	0.894	14
784x140x10	0.978	0.888	14
784x250x10	0.967	0.874	9

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.977	0.883	8
784x80x10	0.974	0.869	12
784x100x10	0.978	0.892	11
784x120x10	0.979	0.894	8
784x140x10	0.978	0.888	11
784x250x10	0.987	0.874	9

Hyperparameters used for tests below:

- for the R-prop $\eta^+ = 1.3, \eta^- = 0.5$
- batch size 1 (online apporach)
- threshold inside GL $\alpha = 1.0$
- strip size and threshold for PQ disjoint and PQ overlap $stripSize = 5, \alpha = 0.5$

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.977	0.883	8
784x80x10	0.978	0.889	5
784x100x10	0.98	0.902	5
784x120x10	0.977	0.884	8
784x140x10	0.977	0.886	5
784x250x10	0.975	0.915	6

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.977	0.883	19
784x80x10	0.977	0.886	9
784x100x10	0.98	0.902	9
784x120x10	0.977	0.884	9
784x140x10	0.977	0.886	9
784x250x10	0.975	0.915	19

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.977	0.883	12
784x80x10	0.977	0.886	9
784x100x10	0.98	0.902	6
784x120x10	0.975	0.877	6
784x140x10	0.977	0.886	9
784x250x10	0.975	0.915	7

Hyperparameters used for tests below:

- for the R-prop $\eta^+ = 1.3, \eta^- = 0.5$
- minibatch number 10
- threshold inside GL $\alpha = 1.0$
- strip size and threshold for PQ disjoint and PQ overlap $stripSize = 5, \alpha = 0.75$

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.976	0.881	4
784x80x10	0.979	0.893	8
784x100x10	0.973	0.864	2
784x120x10	0.974	0.87	7
784x140x10	0.98	0.901	5
784x250x10	0.979	0.895	5

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.979	0.897	14
784x80x10	0.979	0.893	24
784x100x10	0.977	0.887	24
784x120x10	0.976	0.878	39
784x140x10	0.982	0.909	29
784x250x10	0.979	0.895	14

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.979	0.897	14
784x80x10	0.979	0.893	18
784x100x10	0.977	0.887	12
784x120x10	0.975	0.876	16
784x140x10	0.982	0.91	18
784x250x10	0.979	0.895	14

Hyperparameters used for tests below:

- for the R-prop $\eta^+ = 1.3, \eta^- = 0.5$
- minibatch number 10
- threshold inside GL $\alpha = 3.0$
- strip size and threshold for PQ disjoint and PQ overlap $stripSize = 5, \alpha = 1.0$

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.976	0.881	7
784x80x10	0.975	0.874	4
784x100x10	0.978	0.889	7
784x120x10	0.975	0.876	6
784x140x10	0.978	0.888	5
784x250x10	0.979	0.894	5

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.977	0.886	39
784x80x10	0.977	0.883	24
784x100x10	0.978	0.89	34
784x120x10	0.975	0.877	24
784x140x10	0.978	0.892	19
784x250x10	0.98	0.899	14

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.976	0.881	12
784x80x10	0.977	0.885	15
784x100x10	0.978	0.89	31
784x120x10	0.975	0.877	18
784x140x10	0.978	0.892	19
784x250x10	0.98	0.899	14

Hyperparameters used for tests below:

- for the R-prop $\eta^+ = 1.3, \eta^- = 0.5$
- minibatch number 10
- threshold inside GL $\alpha = 5.0$
- strip size and threshold for PQ disjoint and PQ overlap $stripSize = 5, \alpha = 2.0$

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.981	0.905	7
784x80x10	0.978	0.889	6
784x100x10	0.978	0.892	5
784x120x10	0.979	0.897	6
784x140x10	0.974	0.871	8
784x250x10	0.979	0.894	4

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.98	0.899	84
784x80x10	0.978	0.891	19
784x100x10	0.98	0.9	194
784x120x10	0.979	0.894	24
784x140x10	0.975	0.873	199
784x250x10	0.981	0.906	69

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.98	0.899	81
784x80x10	0.978	0.891	19
784x100x10	0.979	0.896	21
784x120x10	0.979	0.894	24
784x140x10	0.975	0.873	81
784x250x10	0.981	0.906	46

Hyperparameters used for tests below:

- for the R-prop $\eta^+ = 1.3, \eta^- = 0.5$
- minibatch number 10
- threshold inside GL $\alpha = 20.0$
- strip size and threshold for PQ disjoint and PQ overlap $stripSize = 5, \alpha = 3.0$

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.979	0.893	8
784x80x10	0.978	0.892	14
784x100x10	0.979	0.897	20
784x120x10	0.98	0.899	10
784x140x10	0.98	0.899	13
784x250x10	0.979	0.894	4

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.98	0.902	154
784x80x10	0.979	0.894	64
784x100x10	0.978	0.889	None
784x120x10	0.98	0.9	64
784x140x10	0.981	0.906	None
784x250x10	0.981	0.905	84

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch
784x60x10	0.98	0.902	154
784x80x10	0.978	0.892	21
784x100x10	0.978	0.89	161
784x120x10	0.98	0.9	64
784x140x10	0.981	0.906	72
784x250x10	0.981	0.905	84

The number of neurons that achieved - in our opinion - the best tradeoff was single layer with 100 nodes. A major percentage predicted can be observed augmenting this number (tested up to 250) but each increase brings a significative increment in time of computation. A little increment in the F-score for a 10x time of computational effort could be an acceptable tradeoff in particular usages.

For the three early stopping criteria tested there is not a clear winner: GL was useful for all the α considered, meanwhile PQ with $\alpha \geq 3.0$ resulted in a much less useful criteria spanning over much more epochs both for disjoint and overlap cases. All of them scored good early stopping, with meaningful choices of α, with high F-score values.

6.3 Rprop Tests

We consider $\eta^+ \in [1.2, 1.8], step = 0.1$ with a single layer 100 nodes Neural Network. We impose for GL and PQ early stopping criteria the α values of 1.0 and 0.5 respectively, and strip size for PQ of 5.

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch	η^+	η^-
784x100x10	0.975	0.877	4	1.2	0.5
784x100x10	0.977	0.887	4	1.3	0.5
784x100x10	0.982	0.91	8	1.4	0.5
784x100x10	0.98	0.902	4	1.5	0.5
784x100x10	0.983	0.913	6	1.6	0.5
784x100x10	0.978	0.892	3	1.7	0.5
784x100x10	0.981	0.904	5	1.8	0.5

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch	η^+	η^-
784x100x10	0.977	0.886	10	1.2	0.5
784x100x10	0.98	0.899	8	1.3	0.5
784x100x10	0.983	0.916	16	1.4	0.5
784x100x10	0.981	0.907	9	1.5	0.5
784x100x10	0.983	0.915	24	1.6	0.5
784x100x10	0.982	0.91	16	1.7	0.5
784x100x10	0.98	0.898	13	1.8	0.5

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch	η^+	η^-
784x100x10	0.977	0.886	14	1.2	0.5
784x100x10	0.98	0.899	9	1.3	0.5
784x100x10	0.983	0.913	24	1.4	0.5
784x100x10	0.981	0.907	9	1.5	0.5
784x100x10	0.983	0.915	24	1.6	0.5
784x100x10	0.981	0.907	24	1.7	0.5
784x100x10	0.979	0.897	19	1.8	0.5

Imposing $\eta^+ = 1.6$ we vary $\eta^- \in [0.3, 0.7], step = 0.1$.

GL

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch	η^+	η^-
784x100x10	0.978	0.888	5	1.6	0.3
784x100x10	0.981	0.907	6	1.6	0.4
784x100x10	0.98	0.898	6	1.6	0.5
784x100x10	0.982	0.912	5	1.6	0.6
784x100x10	0.978	0.89	6	1.6	0.7

PQ Overlap

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch	η^+	η^-
784x100x10	0.98	0.9	14	1.6	0.3
784x100x10	0.981	0.907	21	1.6	0.4
784x100x10	0.983	0.913	15	1.6	0.5
784x100x10	0.982	0.912	16	1.6	0.6
784x100x10	0.978	0.89	9	1.6	0.7

PQ Disjoint

Net Configuration	$Accuracy_{AVG}$	$F\text{-}score_{\mu}$	Stop Epoch	η^+	η^-
784x100x10	0.98	0.9	14	1.6	0.3
784x100x10	0.981	0.907	24	1.6	0.4
784x100x10	0.98	0.902	54	1.6	0.5
784x100x10	0.982	0.912	29	1.6	0.6
784x100x10	0.978	0.89	9	1.6	0.7

In conclusion, for the task of classification of handwritten digits, we would advice a net configuration with 100 neurons in the (single) hidden layer, as stopping criteria PQ Overlap (with $\alpha = 0.5$ and $stripSize = 5$) resilient back propagation parameters as $\eta^+ = 1.6$ and $\eta^- = 0.6$.

7 Python code for Neural Network

7.1 Main.py

```
1     # -*- coding: utf-8 -*-
2 import matplotlib.pyplot as plt
3 import MNIST_Loader as loader
```

```
4  import Function as f
5  import numpy as np
6  import Utility as myU
7  from NeuralNetwork import Net
8
9  pathTrainingImages = 'mnist/train-images-idx3-ubyte'
10 pathTrainingLabels = 'mnist/train-labels-idx1-ubyte'
11 pathTestImages = 'mnist/t10k-images-idx3-ubyte'
12 pathTestLabels = 'mnist/t10k-labels-idx1-ubyte'
13
14 trainingSet_Images = loader.loadImages(pathTrainingImages) # X independent↩
       variable
15 trainingSet_Labels = loader.loadLabels(pathTrainingLabels) # Y dependent ↩
      variable
16 testSet_Images = loader.loadImages(pathTestImages)
17 testSet_Labels = loader.loadLabels(pathTestLabels)
18
19 trainingSet_Images = myU.normalize(trainingSet_Images)
20 testSet_Images = myU.normalize(testSet_Images)
21
22 activationFunctionFirstLayer = f.getSigmoid()
23 activationFunctionOutputLayer = f.getSoftmax_forCrossEntropy()
24 errorFunction = f.getCrossEntropy_forSoftmax()
25
26 # ONLY FOR TESTING PURPOSE.
27 #neuronsForLayerList = [[60,  10], [80,  10], [100,  10], [120,  10], ↩
      [140,  10], [250, 10]] # <------ change neurons number here
28 neuronsForLayerList = [[100,  10]]
29 layersNumber = len(neuronsForLayerList[0])
30 activationFunctionsLayer = [activationFunctionFirstLayer, ↩
      activationFunctionOutputLayer]
31
32 metricsForEachNets = []
33 for h in range(len(neuronsForLayerList)):
34     neuronsForLayer = neuronsForLayerList[h]
35     bias = []
36     metricsForEachNets.append({})
37     metricsForEachNets[h]['netConfiguration'] = ' 784'
38
39     for i in range(layersNumber):
40         bias.append(np.ones(neuronsForLayer[i]))
41         metricsForEachNets[h]['netConfiguration'] = metricsForEachNets[h][↩
               'netConfiguration'] +'x'+ str(neuronsForLayer[i])
42     myNet = Net(layersNumber, neuronsForLayer, activationFunctionsLayer, ↩
          errorFunction, bias)
43     stoppingCriteriaList = myNet.build(trainingSet_Images, ↩
          trainingSet_Labels, epochs=200, trainTo=10000, minibatchNumber=10)
```

```
44
45      for stoppingCriteria in stoppingCriteriaList:
46          stoppingCriteriaName = stoppingCriteria.__class__.__name__
47          metricsForEachNets[h][stoppingCriteriaName] = myNet.↩
                evaluateStoppingCriteria(testSet_Images, testSet_Labels, ↩
                stoppingCriteria, testTo=1000)
48
49  beginCenter = '\\begin{center}\n'
50  beginTabular = '    \\begin{tabular}{||c c c c c c||}\n'
51  hline = '\\hline\n'
52  columnsName = '        Net Configuration & AVG Accuracy & Micro F-Score & ↩
        Stop Epoch & etaPlus & etaMin $\n'
53  endTabular = '    \\end{tabular}\n'
54  endCenter = '\\end{center}\n'
55
56  for stoppingCriteriaName in ['_GL', '_PQ_disjoint', '_PQ_overlap']:
57      File_object = open(stoppingCriteriaName + ".json", "a")
58      File_object.write(beginCenter)
59      File_object.write(beginTabular)
60      File_object.write(hline)
61      File_object.write(columnsName)
62      File_object.write('\\\\ [0.5ex]\n')
63      File_object.write(hline)
64      File_object.close()
65
66  for metric in metricsForEachNets:
67      for stoppingCriteriaName in ['_GL', '_PQ_disjoint', '_PQ_overlap']:
68          resultList = metric.get(stoppingCriteriaName)
69          File_object = open(stoppingCriteriaName+".json", "a")
70
71          File_object.write(hline)
72          row = '        '
73          row = row + metric.get('netConfiguration') + ' & '
74
75          for i in range(len(resultList)):
76              if(i < len(resultList)-1 and i > 0):
77                  row = row + str(np.round(resultList[i], decimals=3)) + ' &↩
                        '
78              elif (i == len(resultList)-1):
79                  row = row + str(resultList[i]) + '  '
80          row = row + '1.6 & 0.6 \\\\'
81          row = row + '[1ex]\n'
82          File_object.write(row)
83          File_object.close()
84
85  for stoppingCriteriaName in ['_GL', '_PQ_disjoint', '_PQ_overlap']:
86      File_object = open(stoppingCriteriaName + ".json", "a")
```

```
87      File_object.write(hline)
88      File_object.write(endTabular)
89      File_object.write(endCenter)
90      File_object.close()
```

7.2 NeuralNetwork.py

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   import StopPredicate as sp
4   import Utility as myU
5   from Function import sign
6   from Layer import Layer
7   from evaluationMetrics import getEvaluationMetrics
8
9   """
10  IMPORTANT: Every array and matrix are treated as numpy array
11  """
12
13  # class Net
14  class Net:
15      layers = []
16      errorFunction = None
17      trainCumulativeErrorForEachEpoch = []
18      validationCumulativeErrorForEachEpoch = []
19      etaPlus = 1.6
20      etaMin = 0.6
21      def __init__(self, layersNumber, neuronsForLayer, activationFunctions,↩
            errorFunction, b):
22          """
23          :param layersNumber: number of layers
24          :param neuronsForLayer: array of numbers
25          :param activationFunctions: array of functions
26          :param errorFunction:
27          :param b: matrix of numbers
28          """
29          self.trainCumulativeErrorForEachEpoch = []
30          self.validationCumulativeErrorForEachEpoch = []
31          self.layers = []
32          self.errorFunction = errorFunction
33          myU.setClassNumber(neuronsForLayer[layersNumber-1])
34          for i in range(layersNumber):
35              b[i] = np.ones(len(b[i]))
36              self.layers.append(Layer(neuronsForLayer[i], ↩
```

```
                activationFunctions[i], b[i]))
37
38      def build(self, X_train, Y_train, epochs=200, trainFrom=0, trainTo=5000, trainStep=1, minibatchNumber=10, validationPercentage=20):
39          self.addWeightsToNeurons(X_train)
40          return self.train(X_train, Y_train, epochs, trainFrom, trainTo, trainStep, minibatchNumber, validationPercentage)
41
42      def train(self, X_train, Y_train, epochs=200, trainFrom=0, trainTo=10000, trainStep=1, minibatchNumber=10, validationPercentage=20):
43          # Take 20% of training set and recalculate trainFrom
44          rangeValidation = int(((trainTo - trainFrom) * validationPercentage)/100)
45          validationFrom, validationTo,  = trainFrom, trainFrom + rangeValidation
46          validationStep, trainFrom = trainStep,  validationTo
47          sizeTrainingSet = myU.numberOfIterations(trainFrom, trainTo, trainStep)
48
49          minibatchSize = np.floor(sizeTrainingSet/minibatchNumber)
50          # Declaration of stopping criteria objects and boolean variables
51          gl, pqDisjoint, pqOverlap = sp.getGl(1.0), sp.getPq_disjoint(strip=5, alpha=0.5), sp.getPq_overlap(strip=5, alpha=0.5)
52          glEarlyStop, pqDisjointEarlyStop, pqOverlapEarlyStop = False, False, False
53
54          labels_count = np.zeros(myU.classNumber)
55          e = 0
56          while e<epochs and (not(glEarlyStop) or not(pqDisjointEarlyStop) or not(pqOverlapEarlyStop)):
57              print("epoch ", e, "/", epochs)
58              #shuffle randomly the training set
59              X_train, Y_train = myU.shufflePairedSet(X_train, Y_train)
60
61              minibatchElementsCounter = 0 # number of elements considered, when a threshold is reached (e.g all the minibatch has been analyzed), update
62              trainLocalCumulativeError = 0
63              for i in range(trainFrom, trainTo, trainStep):
64                  input = myU.getInputAsMonodimensional(X_train[i])
65                  label = myU.getLabelVector(Y_train[i])
66                  labels_count[Y_train[i]] += 1.0
67                  # calculate z vector for each layer
68                  self.forwardPropagation(X_train[i])
69                  # calculate delta vector for each layer
70                  self.backwardPropagation(X_train[i], label)
71                  # for each neuron inside a layer calculate of weights
```

```
                derivative of error function and sum to the previous ↩
                derivative (useful for updating weights)
72              self.sumDerivative(input)
73              minibatchElementsCounter+=1
74              # update weights if it's reached the end of batch
75              if minibatchElementsCounter >= minibatchSize:
76                  self.update()
77                  minibatchElementsCounter=0
78              # calculate error on single sample
79              result = self.layers[len(self.layers) - 1].output
80              trainError = self.errorFunction.calculate(result, label)
81              # accumulate errors on singles samples
82              trainLocalCumulativeError = trainLocalCumulativeError + ↩
                    trainError
83          # evaluate error on the whole training set
84          self.trainCumulativeErrorForEachEpoch.append(↩
                trainLocalCumulativeError/sizeTrainingSet) #used to plot ↩
                error
85          # evaluate error on the whole validation set
86          self.calculateErrorOnValidationSet(validationFrom, ↩
                validationTo, validationStep, X_train, Y_train)
87          validationLocalCumulativeError = self.↩
                validationCumulativeErrorForEachEpoch[-1]
88          """Definition of stop criteria"""
89          if not(glEarlyStop):
90              if validationLocalCumulativeError < gl.getE_opt():
91                  self.remeberThisNet(gl, e)
92              glEarlyStop = gl.shouldEarlyStop(↩
                    validationLocalCumulativeError, e)
93              #if(glEarlyStop): print("GL stops at ", e)
94          if not(pqDisjointEarlyStop):
95              if validationLocalCumulativeError < pqDisjoint.getE_opt():
96                  self.remeberThisNet(pqDisjoint, e)
97              pqDisjointEarlyStop = pqDisjoint.shouldEarlyStop(↩
                    trainLocalCumulativeError, ↩
                    validationLocalCumulativeError, e)
98              #if(pqDisjointEarlyStop): print("PQ_disjoint stops at ", e↩
                    )
99          if not(pqOverlapEarlyStop):
100             if validationLocalCumulativeError < pqOverlap.getE_opt():
101                 self.remeberThisNet(pqOverlap, e)
102             pqOverlapEarlyStop = pqOverlap.shouldEarlyStop(↩
                    trainLocalCumulativeError, ↩
                    validationLocalCumulativeError, e)
103             #if(pqOverlapEarlyStop): print("PQ_overlap stops at ", e)
104         e += 1
105
```

```
106         lastEndingCriteria = np.argmin(np.asarray([gl.getE_opt(), ↩
                pqDisjoint.getE_opt(), pqOverlap.getE_opt()]))
107         lastEndingCriteria = [gl, pqDisjoint, pqOverlap][↩
                lastEndingCriteria]
108         self.updateNetThroughBestWeights(lastEndingCriteria)
109 
110         return [gl, pqDisjoint, pqOverlap]
111 
112     def test(self, X_test, Y_test, testFrom=0, testTo=1000, testStep=1, ↩
            monitorEveryTest=False):
113         truePositives = np.zeros(myU.classNumber)
114         falseNegatives = np.zeros(myU.classNumber)
115         falsePositives = np.zeros(myU.classNumber)
116         for i in range(testFrom, testTo, testStep):
117             result = self.vectorialPrediction(X_test[i])
118             interpreted = np.argmax(result)
119             if monitorEveryTest:
120                 print("Test su Img ", i, "(", Y_test[i], "): ", result)
121                 print("..expected, ", Y_test[i], "; interpreted as ", ↩
                        interpreted, " with ", result[interpreted], "\n")
122             if interpreted != Y_test[i]:
123                 if monitorEveryTest:
124                     print("while P(", Y_test[i], ") was ", result[Y_test[i↩
                            ]])
125                 falseNegatives[Y_test[i]] += 1.
126                 falsePositives[interpreted] += 1.
127             else:
128                 truePositives[Y_test[i]] += 1.
129         print("number of correct predictions(TP) ", np.sum(truePositives),↩
                " --> ", truePositives)
130         print("number of Missed predictions(FN) ", np.sum(falseNegatives),↩
                " --> ", falseNegatives)
131         print("number of Mismatched predictions(FP) ", np.sum(↩
                falsePositives), " --> ", falsePositives)
132         testSize = myU.numberOfIterations(testFrom, testTo, testStep)
133         evaluationMetrics = getEvaluationMetrics(truePositives, ↩
                falsePositives, falseNegatives, testSize)
134         print("testSize ", testSize)
135         print("global guessRate ", evaluationMetrics.getGuessRate())
136         print("Accuracy: ", evaluationMetrics.getAccuracy())
137         print("Precision: ", evaluationMetrics.getPrecision())
138         print("Recall: ", evaluationMetrics.getRecall())
139         print("F-Score: ", evaluationMetrics.getFScore())
140         print("AVG Accuracy: ", evaluationMetrics.getAVGAccuracy())
141         print("Micro Precision: ", evaluationMetrics.getMicroPrecision())
142         print("Micro Recall: ", evaluationMetrics.getMicroRecall())
143         print("Micro F-Score: ", evaluationMetrics.getMicroFScore())
```

```
144         #self.plotError()
145         return [testSize, evaluationMetrics.getGuessRate(), ↩
                evaluationMetrics.getAVGAccuracy(), evaluationMetrics.↩
                getMicroFScore()]
146
147     def evaluateStoppingCriteria(self, X_test, Y_test, stoppingCriteria, ↩
            testFrom=0, testTo=1000, testStep=1):
148         self.updateNetThroughBestWeights(stoppingCriteria)
149         resultList = self.test(X_test, Y_test, testFrom, testTo, testStep)
150         resultList.append(stoppingCriteria.getStopEpoch())
151         return resultList
152
153     def addWeightsToNeurons(self, X_train):
154         numberOfWeightsForNeuron = 0
155         for i in range(len(self.layers)):
156             if i == 0:
157                 numberOfWeightsForNeuron = len(X_train[0]) * len(X_train↩
                        [0][0])
158             else:
159                 numberOfWeightsForNeuron = self.layers[i - 1].↩
                        getNeuronsNumber()
160             self.layers[i].createWeightsMatrix(numberOfWeightsForNeuron)
161
162     def forwardPropagation(self, x):
163         x = np.concatenate(x, axis=0)  # convert x to monodimensional ↩
                array
164         self.layers[0].activate(x)
165         for i in range(1, len(self.layers)):
166             input = self.layers[i-1].output
167             self.layers[i].activate(input)
168
169     def backwardPropagation(self, x, label):
170         #last layer delta is calculated here and memorized in the layers
171         lastLayer = self.layers[len(self.layers) - 1]
172         lastLayer.delta_k = np.asarray(self.errorFunction.↩
                calculateDerivative(lastLayer.output, label)) *lastLayer.↩
                activationFunction.calculateDerivative(lastLayer.a)
173         for i in range( len(self.layers) -2 , -1, -1):
174             nextLayer = self.layers[i + 1]
175             weights_without_bias = np.delete(nextLayer.weights, np.size(↩
                    nextLayer.weights[0]) - 1, 1)
176             self.layers[i].delta_k = np.dot( nextLayer.delta_k, ↩
                    weights_without_bias )  * self.layers[i].↩
                    activationFunction.calculateDerivative(self.layers[i].a)
177         return self
178
179     def sumDerivative(self, input):
```

```
180         for j in range(len(self.layers)):
181             old_derivate = self.layers[j].derivate
182             self.layers[j].derivate = self.layers[j].calculateDerivate(↩
                    input) + old_derivate
183             input = self.layers[j].output
184 
185     def update(self):
186         for i in range(len(self.layers)):
187             layer = self.layers[i]
188             layer.updateValue, updateWeight, layer.previousDerivate = self↩
                    .calculateUpdateValue(layer.derivate, layer.↩
                    previousDerivate, layer.updateValue, layer.↩
                    previousWeightUpdate)
189             layer.weights = layer.weights + updateWeight
190             layer.previousWeightUpdate = updateWeight
191             layer.derivate = np.zeros(np.size(layer))
192         return self
193 
194     def calculateErrorOnValidationSet(self, validationFrom, validationTo, ↩
            validationStep, X_train, Y_train):
195         validationLocalCumulativeError = 0
196         sizeValidationSet = myU.numberOfIterations(validationFrom, ↩
                validationTo, validationStep)
197         for k in range(validationFrom, validationTo, validationStep):
198             predicted = self.vectorialPrediction(X_train[k])
199             label = myU.getLabelVector(Y_train[k])
200             validationError = self.errorFunction.calculate(predicted, ↩
                    label)
201             validationLocalCumulativeError = ↩
                    validationLocalCumulativeError + validationError
202         self.validationCumulativeErrorForEachEpoch.append(↩
                validationLocalCumulativeError/sizeValidationSet)
203 
204     def calculateUpdateValue(self, currentDerivateError, ↩
            previousDerivateError, updateValue, previousUpdateWeight, ↩
            updateMin = 1e-6 , updateMax = 1.0 ):
205         currUpdateValue = np.zeros((len(currentDerivateError), len(↩
                currentDerivateError[0])))
206         currUpdateWeight = np.zeros((len(currentDerivateError), len(↩
                currentDerivateError[0])))
207         for i in range(len(currentDerivateError)):
208             for j in range(len(currentDerivateError[i])):
209                 if(currentDerivateError[i][j] * previousDerivateError[i][j↩
                        ] > 0):
210                     currUpdateValue[i][j] = min(updateValue[i][j] * self.↩
                            etaPlus, updateMax)
211                     currUpdateWeight[i][j] = -1 * sign(↩
```

```
                        currentDerivateError[i][j]) * currUpdateValue[i][j]
                        ]
212                 elif(currentDerivateError[i][j] * previousDerivateError[i
                    ][j] < 0 ):
213                     currUpdateValue[i][j] = max(updateValue[i][j] * self.
                        etaMin, updateMin)
214                     currUpdateWeight[i][j] = -1 * previousUpdateWeight[i][
                        j]
215                     currentDerivateError[i][j] = 0.0
216                 else:
217                     currUpdateValue[i][j] = updateValue[i][j]
218                     currUpdateWeight[i][j] = -1 * sign(
                        currentDerivateError[i][j]) * currUpdateValue[i][j]
                        ]
219
220         return currUpdateValue, currUpdateWeight, currentDerivateError
221
222     def plotError(self):
223         numberOfEpochs = len(self.trainCumulativeErrorForEachEpoch)
224         epochs = np.arange(numberOfEpochs)
225         plt.plot(epochs, self.trainCumulativeErrorForEachEpoch, color='
            blue', label="Train Error")
226         plt.plot(epochs, self.validationCumulativeErrorForEachEpoch, color
            ='green', label="Validation Error")
227         plt.ylabel('Errors values')
228         plt.xlabel('Numbers of Epochs')
229         plt.suptitle('Error trend')
230         plt.legend()
231         plt.show()
232
233     def vectorialPrediction(self, x):
234         self.forwardPropagation(x)
235         return self.layers[len(self.layers) - 1].output
236
237     def predict(self, x):
238         predict = self.vectorialPrediction(x)
239         return np.argmax(predict)
240
241     def remeberThisNet(self, stoppingCriteria, e):
242         layersNumber = len(self.layers)
243         bestConfiguration = []
244         for i in range(layersNumber):
245             bestConfiguration.append(self.layers[i].weights)
246         stoppingCriteria.setOptWheights(bestConfiguration, e)
247
248     def updateNetThroughBestWeights(self, stoppingCriteria):
249         layersNumber = len(self.layers)
```

```
250         bestConfiguration = stoppingCriteria.getOptWheights()
251         for i in range(layersNumber):
252             layer = self.layers[i]
253             numberOfWeightsForNeuron = len(layer.weights[0])
254             neuronsNumber = layer.neuronsNumber
255 
256             for r in range(neuronsNumber):
257                 for c in range(numberOfWeightsForNeuron):
258                     layer.weights[r][c] = bestConfiguration[i][r][c]
```

7.3 Layer.py

```
1     import random
2 import numpy as np
3 
4 class Layer:
5     weights = []
6     output = []
7     a = []
8     b = []
9     activationFunction = None
10     neuronsNumber = 0
11     delta_k = []
12     derivate = []
13     previousDerivate = []
14     updateValue = []
15     previousWeightUpdate = []
16 
17     def __init__(self, neuronsNumber, activationFunction, b):
18         self.weights = []
19         self.derivate = []
20         self.previousDerivate = []
21         self.updateValue = []
22         self.previousWeightUpdate = []
23         self.activationFunction = activationFunction
24         self.neuronsNumber = neuronsNumber
25         self.b = np.asarray(b).reshape((1, len(b)))
26         self.output = np.zeros(neuronsNumber)
27         self.a = np.zeros(neuronsNumber)
28         self.delta_k = np.zeros(neuronsNumber)
29 
30     def createWeightsMatrix(self, numberOfWeightsForNeuron):
31         self.weights = np.zeros((self.neuronsNumber, ↩
               numberOfWeightsForNeuron + 1))
```

```
32          self.derivate = np.zeros((self.neuronsNumber, ↩
                numberOfWeightsForNeuron + 1))
33          self.previousDerivate = np.zeros((self.neuronsNumber, ↩
                numberOfWeightsForNeuron + 1))
34          self.updateValue = 0.1 * np.ones((self.neuronsNumber, ↩
                numberOfWeightsForNeuron + 1))
35          self.previousWeightUpdate = 0.0 * np.ones((self.neuronsNumber, ↩
                numberOfWeightsForNeuron + 1))
36
37          for r in range(self.neuronsNumber):
38              for c in range(numberOfWeightsForNeuron):
39                  weight = random.uniform(0.05, 0.99)
40                  if random.choice([True, False]):
41                      weight *= -1
42                  self.weights[r][c] = weight
43              self.weights[r][numberOfWeightsForNeuron] = self.b[0][r]
44
45      def activate(self, x):
46          self.a = None
47          self.output = None
48          x_b = np.append(x, 1.)
49          self.a = np.asarray(np.dot(x_b, self.weights.T))
50          self.output = self.activationFunction.calculate(self.a)
51          return self.output
52
53      def calculateDerivate(self, z):
54          z = np.append(z, 1.)
55          z = z.reshape(1, np.size(z))
56          dot = np.dot(self.delta_k.T, z)
57          self.derivate = dot
58          return self.derivate
59
60      def getNeuronsNumber(self):
61          return self.neuronsNumber
62
63      def getActivationFunction(self):
64          return self.activationFunction
65
66      def getWeights(self):
67          return self.weights
68
69      def addBiasToWeights(self):
70          self.weights = np.append(self.weights, self.b, axis=1)
```

7.4 evaluationMetrics.py

```
    import numpy as np

class EvaluationMetrics:
    accuracy = []
    precision = []
    recall = []
    fScore = []

    def __init__(self, truePositives, falsePositives, falseNegatives, testSize, beta):
        trueNegatives = testSize-(truePositives+falsePositives+falseNegatives)
        self.guessRate = np.sum(truePositives)/testSize
        self.accuracy = (truePositives+trueNegatives)/testSize
        self.precision = truePositives/(truePositives+falsePositives)
        self.recall = truePositives/(truePositives+falseNegatives)
        self.fScore = ( beta*beta + 1 )* self.precision*self.recall / (beta*beta*self.precision + self.recall)

        classes = np.size(truePositives)
        self.avgAccuracy = np.sum(self.accuracy)/classes
        self.macroPrecision = np.sum(self.precision)/classes
        self.macroRecall = np.sum(self.recall)/classes
        self.microPrecision = np.sum(truePositives)/np.sum(truePositives+falsePositives)
        self.microRecall = np.sum(truePositives)/np.sum(truePositives+falseNegatives)
        self.microFScore = ( beta*beta + 1 )* self.microPrecision*self.microRecall / (beta*beta*self.microPrecision + self.microRecall)
        self.macroFScore = ( beta*beta + 1 )* self.macroPrecision*self.macroRecall / (beta*beta*self.macroPrecision + self.macroRecall)

    def getGuessRate(self):
        return self.guessRate

    def getAccuracy(self, of=None):
        if of is None:
            return self.accuracy
        else:
            try:
                return self.accuracy[of]
            except:
```

```
36                 return self.accuracy
37
38     def getPrecision(self, of=None):
39         if of is None:
40             return self.precision
41         else:
42             try:
43                 return self.precision[of]
44             except:
45                 return self.precision
46
47     def getRecall(self, of=None):
48         if of is None:
49             return self.recall
50         else:
51             try:
52                 return self.recall[of]
53             except:
54                 return self.recall
55
56     def getFScore(self, of=None):
57         if of is None:
58             return self.fScore
59         else:
60             try:
61                 return self.fScore[of]
62             except:
63                 return self.fScore
64
65     def getAVGAccuracy(self):
66         return self.avgAccuracy
67
68     def getMacroPrecision(self):
69         return self.macroPrecision
70
71     def getMacroRecall(self):
72         return self.macroRecall
73
74     def getMacroFScore(self):
75         return self.macroFScore
76
77     def getMicroPrecision(self):
78         return self.microPrecision
79
80     def getMicroRecall(self):
81         return self.microRecall
82
```

```
83      def getMicroFScore(self):
84          return self.microFScore
85
86  def getEvaluationMetrics(truePositives, falsePositives, falseNegatives, ↩
        testSize, beta=2.0):
87      return EvaluationMetrics(truePositives, falsePositives, falseNegatives↩
            , testSize, beta)
```

7.5 Function.py

```
1   """
2   Module that contains functions useful for a Neural Network
3   ==================================
4
5   Provides
6      Activation Functions
7      1. sigmoid: returns a value between 0 and 1
8      2. heavside: returns 1 if input is grater than 0, 0 otherwise
9      3. identity: returns x lol
10     4. softmax: returns the probability distribution of the layer
11
12     Error Functions
13     1. sumOfSquares: useful for regression (continue)
14     2. crossEntropy: useful for classification (discrete)
15
16     Other Functions
17     1. sign: returns the sign of a value (+1; 0; -1)
18  """
19  import numpy as np
20
21  #getter
22  def getIdentity():
23      if _Identity._instance is None:
24          _Identity._instance = _Identity()
25      return _Identity._instance
26
27  def getSigmoid():
28      if _Sigmoid._instance is None:
29          _Sigmoid._instance = _Sigmoid()
30      return _Sigmoid._instance
31
32  def getHeavside():
33      if _Heavside._instance is None:
34          _Heavside._instance = _Heavside()
```

```
35      return _Heavside._instance
36
37  def getSoftmax():
38      if _Softmax._instance is None:
39          _Softmax._instance = _Softmax()
40      return _Softmax._instance
41
42  def getSoftmax_forCrossEntropy():
43      if _Softmax_forCrossEntropy._instance is None:
44          _Softmax_forCrossEntropy._instance = _Softmax_forCrossEntropy()
45      return _Softmax_forCrossEntropy._instance
46
47  def getSumOfSquares():
48      if _SumOfSquares._instance is None:
49          _SumOfSquares._instance = _SumOfSquares()
50      return _SumOfSquares._instance
51
52  def getCrossEntropy():
53      if _CrossEntropy._instance is None:
54          _CrossEntropy._instance = _CrossEntropy()
55      return _CrossEntropy._instance
56
57  def getCrossEntropy_forSoftmax():
58      if _CrossEntropy_forSoftmax._instance is None:
59          _CrossEntropy_forSoftmax._instance = _CrossEntropy_forSoftmax()
60      return _CrossEntropy_forSoftmax._instance
61
62  # Activation Functions
63  class ActivationFunction:
64      _instance = None
65
66      def calculate(self, x):
67          pass
68
69      def calculateDerivative(self, x):
70          pass
71
72  class _Identity(ActivationFunction):
73      def calculate(self, x):
74          return x  # lol
75
76      def calculateDerivative(self, x):
77          return np.ones((1, x.size))
78
79  class _Sigmoid(ActivationFunction):
80      maxFloat64 = np.finfo(np.float64).max
81      epsFloat64 = np.finfo(np.float64).eps
```

```
82      tendsToOne = 1.0/(1.0+(np.finfo(np.float32).eps/16))
83      smallest = np.nextafter(0, 1)
84      def calculate(self, x):
85          '''return 1.0 / (1.0 + np.exp(-x))'''
86          expWontOverflow = -x < np.log(self.maxFloat64)
87          expWontUnderflow = -x > np.log(self.epsFloat64)
88          return np.where(expWontOverflow,
89                 np.where(expWontUnderflow, 1.0 / (1.0 + np.exp(-x, where=↩
                    expWontOverflow)), self.tendsToOne),
90                 self.smallest)
91
92      def calculateDerivative(self, x):
93          return self.calculate(x) * (1.0 - self.calculate(x))
94
95  class _Heavside(ActivationFunction):
96      def calculate(self, x):
97          return 1.0 if x > 0 else 0.0
98
99      def calculateDerivative(self, x):
100         return np.zeros(x.size)
101
102 class _Softmax(ActivationFunction):
103     def calculate(self, x):
104         #shiftedX = x - np.max(x)
105         exp = np.exp(x)
106         sum = np.sum(exp)
107         return exp/sum
108
109     def calculateDerivative(self, x):
110         ret = ( 1.0 / self.calculate(x) ) - 1.0
111         return ret.reshape(1, len(x))
112
113 class _Softmax_forCrossEntropy(ActivationFunction):
114     def calculate(self, x):
115         exp = np.exp(x)
116         sum = np.sum(exp)
117         return exp/sum
118
119     def calculateDerivative(self, x):
120         return np.ones(np.size(x))
121
122 #Funzioni d'errore
123 class ErrorFunction:
124     _instance = None
125
126     def calculate(self, y, t):
127         pass
```

```
128
129     def calculateDerivative(self, y, t):
130         pass
131
132     def areSameSize(self, y, t):
133         if y.size != t.size:
134             raise Exception("The arguments must have same size!")
135         return True
136
137 class _SumOfSquares(ErrorFunction):
138     def calculate(self, y, t):
139         if self.areSameSize(y, t):
140             diff = y - t
141             result = np.sum(diff**2)
142             return result/2.0
143
144     def calculateDerivative(self, y, t):
145         print("y and t", y, t)
146         if self.areSameSize(y, t):
147             return np.sum(y-t);
148
149 class _CrossEntropy(ErrorFunction):
150     smallest = np.nextafter(0, 1)
151     def calculate(self, y, t):
152         if self.areSameSize(y, t):
153             y= np.where(y==0, self.smallest, y)
154             return -np.sum(t * np.log(y))
155
156     def calculateDerivative(self, y, t):
157         if self.areSameSize(y, t):
158             return np.sum(-t / y)
159
160 class _CrossEntropy_forSoftmax(ErrorFunction):
161     smallest = np.nextafter(0, 1)
162     def calculate(self, y, t):
163         if self.areSameSize(y, t):
164             y= np.where(y==0, self.smallest, y)
165             return -np.sum(t * np.log(y))
166
167     def calculateDerivative(self, y, t):
168         if self.areSameSize(y, t):
169             ret = y - t
170             return ret.reshape(1, len(y))
171
172 def sign(x):
173     return np.where(x > 0, 1, np.where(x < 0, -1, 0))
```

7.6 StopPredicate.py

```python
import numpy as np

def getGl(alpha = 1.0):
    return _GL(alpha)

def getPq_disjoint(strip=5, alpha=0.5):
    return _PQ_disjoint(strip, alpha)

def getPq_overlap(strip=5, alpha=0.5):
    return _PQ_overlap(strip, alpha)

def getNoStopCriteria():
    return NoStop()

def getNaiveFirstIncreaseCriteria():
    return FisrtIncrease()

class StopPredicate:
    def __init__(self):
        self.optWeights = []
        self.epochOfOptWeights = None
        self.stopEpoch = None

    #(NEW) Forse da togliere, che PQ non lo usa
    def shouldEarlyStop(self, x, epoch):
        pass

    def getOptWheights(self):
        return self.optWeights

    def setOptWheights(self, weightsToSet, e):
        self.optWeights=weightsToSet
        self.epochOfOptWeights = e

    def getStopEpoch(self):
        return self.stopEpoch

    def getBestEpoch(self):
        return self.epochOfOptWeights

    def getE_opt(self):
        pass

class NoStop(StopPredicate):
```

```
45      def __init__(self):
46          super(NoStop, self).__init__()
47          self.E_opt = np.inf
48
49      def shouldEarlyStop(self, validationError, epoch):
50          if(validationError<self.E_opt):
51              self.E_opt = validationError
52          return False
53
54      def getE_opt(self):
55          return self.E_opt
56
57  class FisrtIncrease(StopPredicate):
58      def __init__(self):
59          super(FisrtIncrease, self).__init__()
60          self.lastError = np.inf
61
62      def shouldEarlyStop(self, validationError, epoch):
63          if validationError > self.lastError:
64              self.lastError=validationError
65              return False
66          else:
67              self.stopEpoch=epoch
68              return True
69
70      def getE_opt(self):
71          return self.lastError
72
73  class _GL(StopPredicate):
74      def __init__(self, alpha=1.0):
75          super(_GL, self).__init__()
76          self.optError = np.inf
77          self.alpha = alpha
78
79      def shouldEarlyStop(self, validationError, epoch):
80          gl = self.calculateGl(validationError)
81          print("GL value: ", gl)
82          if (gl > self.alpha):
83              self.stopEpoch=epoch
84              return True
85          else:
86              return False
87
88      def calculateGl(self, validationError):
89          self.updateE_opt(validationError)
90          return 100 * ((validationError/self.optError) - 1)
91
```

```
    def updateE_opt(self, validationError):
        if validationError<self.optError:
            self.optError = validationError

    def getE_opt(self):
        return self.optError

class _PQ_disjoint(StopPredicate):

    def __init__(self, stripSize=5, alpha=0.5):
        super(_PQ_disjoint, self).__init__()
        self.stripSize = stripSize
        self.alpha = alpha
        self.curr = 0
        self.trainingErrors = np.zeros(stripSize)
        self.gl = getGl()

    def shouldEarlyStop(self, trainError, validationError, epoch):
        self.addError(trainError, validationError)
        if self.curr != self.stripSize:
            return False
        pk = self.calculatePk(self.trainingErrors)
        glValue = self.gl.calculateGl(validationError)

        self.curr = 0
        print("PQ_disjoint value: ", glValue/pk)
        if (glValue/pk > self.alpha):
            self.stopEpoch=epoch
            return True
        else:
            return False

    def addError(self, trainError, validationError):
        self.trainingErrors[self.curr] = trainError
        self.curr = self.curr + 1
        self.gl.updateE_opt(validationError)

    def calculatePk(self, trainError):
        minError = np.min(trainError)
        cumulativeError = np.sum(trainError)
        return 1000 * ((cumulativeError/(self.stripSize * minError)) - 1)

    def getStrip(self):
        return self.stripSize

    def getE_opt(self):
        return self.gl.optError
```

```
139
140 class _PQ_overlap(StopPredicate):
141
142     def __init__(self, stripSize=5, alpha=0.5):
143         super(_PQ_overlap, self).__init__()
144         self.stripSize = stripSize
145         self.alpha = alpha
146         self.curr = 0
147         self.isFull = False
148         self.trainingErrors = np.zeros(stripSize)
149         self.gl = getGl()
150
151     def shouldEarlyStop(self, trainError, validationError, epoch):
152         self.addError(trainError, validationError)
153         if not(self.isFull):
154             return False
155         pk = self.calculatePk(self.trainingErrors)
156         glValue = self.gl.calculateGl(validationError)
157         print("PQ_overlap value: ", glValue/pk)
158         if (glValue/pk > self.alpha):
159             self.stopEpoch=epoch
160             return True
161         else:
162             return False
163
164     def addError(self, trainingError, validationError):
165         self.trainingErrors[self.curr] = trainingError
166         self.curr = (self.curr + 1) % self.stripSize
167         self.gl.updateE_opt(validationError)
168         self.isFull = self.isFull or (self.curr==0)
169
170     def calculatePk(self, trainingErrors):
171         minTError = np.min(trainingErrors)
172         sumTError = np.sum(trainingErrors)
173         return 1000 * ((sumTError/(self.stripSize * minTError)) - 1)
174
175     def getStrip(self):
176         return self.stripSize
177
178     def getE_opt(self):
179         return self.gl.optError
```

7.7 Utility.py

```
    """
Module that contains a miscellaneous functions
==================================

Provides
   1. extractTsAndVs: returns a 2x[] matrix containing training set and validation set labels
   2. normalize: accept as argument a matrix and returns the same matrix which every value is between 0 and 1
   3. getLabelVector: accept as argument a label in [0; 9] and returns a numpyArray of all zeros except a 1 in position label
   4. getInputAsMonodimensional: converts an image (as matrix) into a monodimensional array
   5. numberOfIterations: given the start, end and step of a set (Training/Test/Validation) and returns the number of iterations
"""
import numpy as np

classNumber = 10

def setClassNumber(numberOfClasses):
    global classNumber
    classNumber = numberOfClasses

def extractTsAndVs(Y, valPercent=0.2):
    global classNumber
    labels = np.unique(Y)
    ind_T = []
    ind_V = []
    index = [ [] for x in range(classNumber) ]
    for i in range(Y.size):
        index[Y[i]].append(i)
    for i in range(classNumber):
        N = len(index[i])
        Nval = int(np.trunc(valPercent*N))
        validationRow = index[i][0:Nval]
        trainingRow = index[i][Nval:]
        ind_V.append(validationRow)
        ind_T.append(trainingRow)

    return [ind_T, ind_V]

def normalize(x, mmin=0.0, mmax=255.0):
    x = (x - mmin )/(mmax - mmin + 10**(-6))
    return x

def shufflePairedSet(firstSet, secondSet):
```

```
43      z = list(zip(firstSet, secondSet))
44      np.random.shuffle(z)
45      return zip(*z)
46
47  def getLabelVector(lab):
48      global classNumber
49      labelVec = np.zeros(classNumber)
50      labelVec[lab] = 1.0
51      return labelVec
52
53  def getInputAsMonodimensional(input):
54      monoDimInput = np.concatenate(input, axis=0)
55      monoDimInput = input.reshape(1, len(monoDimInput))
56      return monoDimInput
57
58  def numberOfIterations(fromIteration, toIteration, stepIteration):
59      return (int)(np.ceil((toIteration-fromIteration)/stepIteration))
```

7.8 MNIST_Loader.py

```
1       # -*- coding: utf-8 -*-
2   """
3   Module that contains a loading from MNIST functions
4   ==================================
5
6   Provides
7      1. loadImages: returns a 28x28x[number of MNIST images] matrix ↩
           containing the raw MNIST images
8      2. loadLabes: returns an array of size [number of MNIST images] ↩
           containing the images labels
9   """
10
11  import struct as st
12  import numpy as np
13
14  def loadImages(filename):
15      try:
16          train_imagesfile = open(filename, "rb")
17
18      except:
19          print("You can't open this file. Please check if the filename is ↩
                correct")
20
21      train_imagesfile.seek(0)
```

```
22      magic = st.unpack('>4B', train_imagesfile.read(4))
23
24      nImg = st.unpack('>I',train_imagesfile.read(4))[0] #num of images
25      nR = st.unpack('>I',train_imagesfile.read(4))[0] #num of rows
26      nC = st.unpack('>I',train_imagesfile.read(4))[0] #num of column
27      nImg = int(nImg/5.5)
28
29      """
30      print("magic: ",magic)
31      print("nImg: ",nImg)
32      print("nR: ",nR)
33      print("nC: ",nC)
34      """
35      images_array = np.zeros((nImg,nR,nC))
36
37      nBytesTotal = nImg*nR*nC*1 #since each pixel data is 1 byte
38      images_array = np.asarray(st.unpack('>'+'B'*nBytesTotal,↩
           train_imagesfile.read(nBytesTotal)))
39      images_array = images_array.reshape((nImg,nR,nC))
40      np.transpose( images_array )
41
42      train_imagesfile.close()
43
44      return images_array
45
46  def loadLabels(filename):
47      try:
48          train_labelsfile = open(filename, "rb")
49
50      except:
51          print("You can't open this file. Please check if the filename is ↩
               correct")
52
53      train_labelsfile.seek(0)
54      magic = st.unpack('>4B', train_labelsfile.read(4))
55
56      nItm = st.unpack('>I',train_labelsfile.read(4))[0] #num of items
57      """
58      print("magic: ",magic)
59      print("nItm: ",nItm)
60      """
61      labels_array = np.zeros(nItm)
62
63      nBytesTotal = nItm*1 #since each pixel data is 1 byte
64      labels_array = np.asarray(st.unpack('>'+'B'*nBytesTotal,↩
           train_labelsfile.read(nBytesTotal)))
65
```

```
66      train_labelsfile.close()
67
68      return labels_array
```

References

[1] L. Prechelt, *Early Stopping - but when?, Springer, Neural Networks: Tricks of the Trade,. 1524: 55-69* (1996).

[2] M. Riedmiller, H. Braun, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. In IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS* (1993).