Computational Mathematics for Data Analysis and Learning

Marinelli Alberto Roberto (638283) - a.marinelli9@studenti.unipi.it
Melero Cavallo Martina (639305) - m.melerocavallo@studenti.unipi.it

Master Degree Computer Science, AI curriculum

Date: 10/02/2023

Abstract

In order to extract the relationships underlying large volumes of data, neural networks have become the predominant choice for machine learning applications. While reasonably good performance can be achieved with *Single Layer Feed-forward Networks* (SLFNs) trained using a classical *Back-Propagation* (BP) approach, these algorithms are still slow in learning and have other disadvantages. Recently, some alternative algorithms have been proposed to address these problems.

In this project it is used a (M) Machine Learning model based on a method called **Extreme Learning** proposed by Huang et al. [3, 4], with the employment of *L2-regularization*. In particular, a comparison was carried out between:

(A1) which is a variant of incremental extreme learning machine (I-ELM) that is *QRI-ELM* [10], which uses **QR factorization** to decompose the matrix of the hidden output layer, reducing the computational complexity involved in computing the Moore-Penrose inverse, and

(A2) which is a **standard momentum descent** (heavy ball) approach, applied to the Extreme Learning Machine.

Contents

1 Introduction

In this report, two algorithms based on **Extreme Learning Machine** (ELM) are analyzed with the aim of studying their characteristics and behavior in the case of convex optimization.

Chapter 2 is dedicated to the description of **ELM** and the related incremental variant (I-ELM) in order to be able to understand their functioning.

The first algorithm is the **QRI-ELM** which uses QR factorization to decompose the matrix of the hidden output layer, reducing the computational complexity involved in computing the Moore-Penrose inverse. Chapter 3 is completely focused on this algorithm and the theoretical concepts needed to understand it.

The second algorithm studied in this report, precisely in Chapter 4, is the **standard momentum descent** (heavy ball) approach applied to the ELM.

Chapter 5 highlights some **optimization features of the implementations** of the above algorithms.

In order to be able to compare the two approaches, using the ML-CUP dataset of the Machine Learning course of the University of Pisa, a series of **experiments** were configured to evaluate the *quality of the solutions*, the *impact of regularization*, the *number of neurons*, and *scalability*. These are documented in Chapter 6 where related results are also reported.

Finally, based on the obtained results, the **final comparison** between the two algorithms is given in the conclusions in Chapter 7.

2 Extreme Learning Machine

Extreme Learning Machine (ELM) is a fast learning algorithm for single hidden layer feedforward neural networks (SLFNs) that allows to overcome several problems that exist in the most popular algorithms like *back-propagation* (BP), also featuring further advantages.

One of the most relevant advantages of using ELM is the learning speed compared to other traditional learning methods. This results from the fact that in ELM the input weights (linking the input layer to the hidden layer) and hidden biases are chosen randomly, and the output weights (linking the hidden layer to the output layer) are analytically determined using the *Moore–Penrose* (MP) pseudo-inverse.

This algorithm has no parameters which need to be tuned except predefined network architecture, this allows to avoid situations where the algorithm convergence is very slow or divergent as, for example, occurs in the approaches based on BP during the regulation of the learning rate η.

The generalization performance of this learning algorithm is often better than gradient-based learning because these can easily get stuck in a local minimum or incur into over-training, so validation and stopping criteria become necessary to avoid these issues.

Despite the various advantages, in ELM the use of the trial and error method is necessary to establish the appropriate number of hidden nodes to correctly solve that specific task. For example, it is shown that the ELM networks are also prone to overfit if the number of hidden nodes increases [3].

Many researchers have developed numerous variants of ELM to further improve the performance. In this project, the basic version of ELM is employed along with the **standard momentum** algorithm. In addition to the original ELM, a variant chosen for this project is one proposed by Huang et al., *incremental extreme learning machine* (I-ELM) [2] to control the number of nodes of the hidden layer. The hidden nodes are added in the hidden layer one by one and the output weights of the existing hidden nodes are frozen when a new hidden node is added. Specifically the variant employed was **QRI-ELM**, which uses **QR factorization** to decompose the pseudo-inverse matrix of the hidden output layer such as $\boldsymbol{H}^{\dagger} = \mathbf{R}^{-1}\mathbf{Q}^{T}$, where $\mathbf{H}$ is the hidden layer output matrix, $\mathbf{Q}$ and $\mathbf{R}$ are an orthogonal and an upper triangular matrix respectively. In this way, this approach simplifies the computation of the pseudo-inverse of hidden layer output matrix ($\mathbf{H}$).

2.1 Single hidden layer feedforward networks with random hidden nodes

Let ($\mathbf{X}$, $\mathbf{T}$) be N samples, $\mathbf{X} = \{x_i[n]\} \in \mathbb{R}^{N \times I}$ each denoting the input data of the i-th input neuron at n-th time instant and $\mathbf{T} = \{t_l[n]\} \in \mathbb{R}^{N \times L}$ each being the target of the

l-th output neuron at n-th time instant; these samples are employed to train a SLFN with I input neurons, K hidden neurons, L output neurons and activation function g$(\cdot)$.

In ELM, the input weights $\{w_{ik}\}$ and hidden biases $\{b_k\}$ are randomly generated; w_{ik} is the weight connecting i-th input neuron to k-th hidden neuron, it incorporates b_k bias of k-th hidden neuron such that $w_{0k} = b_k$ and $x_0[n] = 1$. The hidden layer output matrix $\mathbf{H} = \{h_k[n]\} \in \mathbb{R}^{N \times K}$ can be obtained by:

$$h_k[n] = g\left(\sum_{i=0}^{I} x_i[n] \cdot w_{ik}\right) \tag{1}$$

Let $\boldsymbol{\beta} = \{\beta_{kl}\} \in \mathbb{R}^{K \times L}$ be the matrix of output weights, where β_{kl} denotes the weight connection between the k-th hidden neuron and the l-th output neuron; and $\mathbf{Y} = \{y_l[n]\} \in \mathbb{R}^{N \times L}$ be the matrix of the network's output data, with $y_l[n]$ the output data in the l-th output neuron at the n-th time instant. The following equation can be obtained for the linear output neurons:

$$y_l[n] = \sum_{k=1}^{K} h_k[n] \cdot \beta_{kl} \tag{2}$$

or

$$\mathbf{Y} = \mathbf{H} \cdot \boldsymbol{\beta} \tag{3}$$

Given the hidden-layer output matrix $\mathbf{H}$ and the targets matrix $\mathbf{T}$, to minimize $\|\mathbf{Y} - \mathbf{T}\|_2$, the output weights can be computed by the *minimum norm least-square* (LS) solution of the linear system:

$$\hat{\boldsymbol{\beta}} = \boldsymbol{H}^{\dagger} \cdot \mathbf{T}, \tag{4}$$

where $\boldsymbol{H}^{\dagger}$ is the **Moore–Penrose** generalized inverse of matrix $\mathbf{H}$ [4, 1].

2.2 Algorithm ELM

The ELM learning algorithm for SLFNs comprehends the following steps:

Algorithm ELM: Given a training set $(\mathbf{X}, \mathbf{T})$ with $\mathbf{X} = \{x_i[n]\} \in \mathbb{R}^{N \times I}$ denoting the input data and $\mathbf{T} = \{t_l[n]\} \in \mathbb{R}^{N \times L}$ the targets, with activation function g$(\cdot)$, and hidden neuron number K,
step 1: Assign arbitrary input weights $\mathbf{w}_k$ and bias $b_k = w_{0k}$, $k = 1, ..., K$.

step 2: Calculate the hidden layer output matrix $\mathbf{H}$.
step 3: Calculate the output weight $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} = \boldsymbol{H}^{\dagger}\mathbf{T} \tag{5}$$

where $\mathbf{H}$, $\boldsymbol{\beta}$ and $\mathbf{T}$ are defined as in 2.1.

2.3 Incremental-ELM

The objective of the **Incremental-ELM** is to avoid the search for the number of neurons by trial and error method, calculating it directly during the training phase.

Given a set of training data, a single hidden layer neural network is trained starting with one hidden node. The randomly generated hidden nodes are added in hidden layer one by one by freezing the output weights of the existing hidden nodes. This process is repeated until the maximum number of hidden nodes is reached or the accuracy is lower than a pre-established one.

The variant of **I-ELM** used in this project is the **QRI-ELM**, which uses **QR** factorization to decompose pseudoinverse matrix of hidden output layer such as $\boldsymbol{H}^{\dagger} = \mathbf{R}^{-1}\mathbf{Q}^{T}$, where $\mathbf{H}$ is the hidden layer output matrix.

3 QRI-ELM

In this section, **QRI-ELM**, variant of **I-ELM**, used in this project is described. QRI-ELM uses the thin **QR** factorization to decompose the pseudoinverse matrix of hidden output layer as $\boldsymbol{H}^{\dagger} = \mathbf{R}^{-1}\mathbf{Q}^{T}$, where $\mathbf{H}$ is the hidden layer output matrix, $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix.

Additionaly, to the network, *Tikhonov regularization* has been applied to avoid overfitting.

3.1 QR factorization

Several methods can be used to compute the Moore–Penrose generalized inverse of $\mathbf{H}$. Among these methods there are: orthogonal projection, the orthogonalization method, the iterative method, and singular value decomposition (SVD) [5]. The SVD can be generally used to calculate the Moore–Penrose generalized inverse of $\mathbf{H}$ in all applications, unfortunately, the computational cost of this method is very high ($\approx 13n^3$ if $m \approx n$ or $2mn^2$ if $m \gg n$, where m is the number of rows and n is the number of columns).

The approach used in this project is based on **QR factorization**, the hidden layer output matrix $\mathbf{H}$ can be decomposed as $\mathbf{H} = \mathbf{Q} \cdot \mathbf{R}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix [9]. This method is not as expensive as SVD for square matrices ($\frac{4}{3}n^3$) and for rectangular matrices with $m \gg n$ (m number of rows and n number of columns), similarly to SVD, it is possible to achieve a complexity linear in the largest dimension ($2mn^2$).

3.1.1 Householder reflectors

The principal method for computing QR factorization is through **Householder reflectors**, which are orthogonal and symmetrical matrices of the form $\mathbf{S} = \mathbf{I} - \frac{2}{\|\mathbf{v}\|^2}\mathbf{v}\mathbf{v}^T$ or $\mathbf{S} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^T$ with $\mathbf{u} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$ for every $\mathbf{v} \in \mathbb{R}^n$. Given a vector $\mathbf{x} \in \mathbb{R}^n$, these matrices produce a mirroring of $\mathbf{x}$, which is $\mathbf{Sx}$, on the other side of the hyperplane perpendicular to $\mathbf{v}$. Specifically, $\mathbf{Sx}$ results in a vector

$$\begin{bmatrix} s \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $s = \pm\|\mathbf{x}\|_2$ due to the fact that $\mathbf{S}$ is an orthogonal matrix, meaning it preserves the length of the vector.

The QR factorization of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is then obtained through the multiplication to the left of orthogonal matrices $\mathbf{Q}_j$. At step $j = 1$, $\mathbf{Q}_1$ is equal to the Householder reflector of the first column of $\mathbf{A}$. At steps $j = 2, ..., n$ these are matrices of form

$$\mathbf{Q}_j = \begin{bmatrix} \mathbf{I}_{j-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}j \end{bmatrix} \tag{6}$$

created by stacking $\mathbf{I}$, identity matrix of size $(j-1) \times (j-1)$, and $\mathbf{S}_j$ Householder reflector of a vector containing the entries $j, ..., m$ of the k-th column of A.

This multiplication results in an upper triangular matrix $\mathbf{R}$, since the matrices $\mathbf{Q}_j$ are orthogonal the product can just be rewritten as

$$\begin{gathered} \mathbf{Q}_n \mathbf{Q}_{n-1} ... \mathbf{Q}_1 \mathbf{A} = \mathbf{R} \\ \mathbf{A} = \underbrace{\mathbf{Q}_1^T ... \mathbf{Q}_{n-1}^T \mathbf{Q}_n^T}_{\mathbf{Q}} \mathbf{R} \\ \mathbf{A} = \mathbf{QR} \end{gathered}$$

Matrices $\mathbf{Q}_j$ are also symmetric so the product could have been written as $\mathbf{A} = \mathbf{Q}_1 ... \mathbf{Q}_n \mathbf{R}$ as well. Hence, $\mathbf{A} \in \mathbb{R}^{m \times n}$ has been decomposed in $\mathbf{Q} \in \mathbb{R}^{m \times m}$ and $\mathbf{R} \in \mathbb{R}^{m \times n}$.
If $m > n$ the factorization results in matrices of the following form:

$$\mathbf{Q} = \left[\underbrace{\mathbf{Q}_1}_{n} \middle| \underbrace{\mathbf{Q}_2}_{m-n} \right]$$

$$\mathbf{R} = \left[\begin{array}{c} \mathbf{R}_1 \\ \hline \mathbf{0} \end{array} \right] \begin{array}{l} \} n \\ \} m - n \end{array}$$

The multiplication and the overall factorization can thus be simplified by only computing $\mathbf{Q}_1 \in \mathbb{R}^{m \times n}$, tall-thin with orthonormal columns, and $\mathbf{R}_1 \in \mathbb{R}^{n \times n}$, square upper-triangular. This method is called **Thin QR**.

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} = \left[\begin{array}{c|c} \mathbf{Q}_1 & \mathbf{Q}_2 \end{array} \right] \cdot \left[\begin{array}{c} \mathbf{R}_1 \\ \hline \mathbf{0} \end{array} \right] = \mathbf{Q}_1 \cdot \mathbf{R}_1 + \mathbf{Q}_2 \cdot \mathbf{0} = \mathbf{Q}_1 \cdot \mathbf{R}_1$$

3.1.2 Least squares with QR factorization

In the least squares problem we want to find a vector $\mathbf{x} \in \mathbb{R}^n$ that satisfies $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$. This problem can be solved through QR factorization, as a matter of fact we can rewrite it as

$$\min_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{Ax} - \mathbf{b}\|_2 = \|\mathbf{QRx} - \mathbf{b}\|_2$$

and multiply everything by $\mathbf{Q}^T$, which is an orthogonal matrix and will preserve the 2-norm.

$$\|\mathbf{Q}^T(\mathbf{QRx} - \mathbf{b})\|_2 = \|\mathbf{Rx} - \mathbf{Q}^T\mathbf{b}\|_2$$

By interpreting the matrices $\mathbf{Q}$ and $\mathbf{R}$ as block matrices as described in 3.1.1 the least squares formulation becomes

$$\|\mathbf{Rx} - \mathbf{Q}^T\mathbf{b}\|_2 = \left\| \begin{bmatrix} \mathbf{R}_1 \\ 0 \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{Q}_1^T \\ \mathbf{Q}_2^T \end{bmatrix} \mathbf{b} \right\|_2 = \left\| \begin{bmatrix} \mathbf{R}_1\mathbf{x} - \mathbf{Q}_1^T\mathbf{b} \\ -\mathbf{Q}_2^T\mathbf{b} \end{bmatrix} \right\|_2$$

Given that the term $-\mathbf{Q}_2^T\mathbf{b}$ does not depend on $\mathbf{x}$, $\|\mathbf{Q}_2^T\mathbf{b}\|$ becomes the residual and the focus is set on minimizing the norm of the first block $\mathbf{R}_1\mathbf{x} - \mathbf{Q}_1^T\mathbf{b}$, which means solving $\mathbf{R}_1\mathbf{x} = \mathbf{Q}_1^T\mathbf{b}$.

Basically, the algorithm to solve least squares through QR factorization would have the following steps:

step 1. Compute the QR factorization $\mathbf{A} = \mathbf{QR}$
step 2. Compute $\mathbf{c} = \mathbf{Q}_1^T\mathbf{b}$
step 3. Solve the upper-triangular system $\mathbf{R}_1\mathbf{x} = \mathbf{c}$

If $\mathbf{A}$ has full column rank then $\mathbf{R}_1$ is invertible and the least square problem has a unique solution (the opposite implication is also true, i.e. if $\mathbf{R}_1$ invertible then A has full column rank, hence there is a unique solution to the LS problem).

3.2 Tikhonov regularization

Tikhonov regularization, also known as Ridge regression, is a method which introduces a pentaly term, which is added to discourage solutions with a large norm. High values are penalized, since the regularization tends towards smaller values, this permits to control model complexity and avoid overfitting.

In the least squares problem, Tikhonov regularization is obtained through the following formula (7).

$$\min_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \lambda^2\|\mathbf{x}\|^2 \tag{7}$$

This can be rewritten as in (8), where $\mathbf{Id}$ is the identity matrix.

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left\| \begin{bmatrix} \mathbf{A} \\ \lambda\mathbf{Id} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \right\|_2^2 \tag{8}$$

3.3 Algorithm QRI-ELM

Algorithm QRI-ELM: Given a training set $(\mathbf{X}, \mathbf{T})$ with $\mathbf{X} = \{x_i[n]\} \in \mathbb{R}^{N \times I}$ denoting the input data and $\mathbf{T} = \{t_l[n]\} \in \mathbb{R}^{N \times L}$ the targets, with activation function $\mathrm{g}(\cdot)$, and maximum hidden neuron number K_{max}.
Also, given $\epsilon \in \mathbb{R}$ threshold for accuracy of the results and $\lambda \in \mathbb{R}$ regularization coefficient,

step 1: Assign arbitrary input weights $\mathbf{w}_k \in \mathbb{R}^{M \times k}$ and bias $b_k = w_{0k}$, with $k = 1$, where $M = I + 1$ because of the inclusion of the bias in the weights.

step 2: Calculate the hidden layer output matrix $\mathbf{H} \in \mathbb{R}^{N \times k}$ as in 1.

step 3: Add regularization coefficient to matrix $\mathbf{H}$, where $\mathbf{Id} \in \mathbb{R}^{k \times k}$ identity matrix, to generate matrix $\mathbf{E} \in \mathbb{R}^{(N+k) \times k}$. Matrix $\mathbf{E}$ is the new matrix to factor.

$$\mathbf{E} = \begin{bmatrix} \mathbf{H} \\ \lambda \mathbf{Id} \end{bmatrix}$$

step 4: Compute Householder vector of $\mathbf{E}$, $\mathbf{u} \in \mathbb{R}^{(N+1) \times 1}$, and store it in matrix $\mathbf{U} \in \mathbb{R}^{(N+1) \times k}$ and compute the value $s = \pm \|\mathbf{x}\|_2$ as stated in 3.1.1, where $\mathbf{x}$ is the k-th column of $\mathbf{E}$. The choice of this value is explained in 5.1.

step 5: matrix $\mathbf{R} \in \mathbb{R}^{k \times k}$ is initialized as $\mathbf{R} = s$ since, in this step, $k = 1$. Further explanations are given in 5.1.

step 6: Compute $\mathbf{c} \in \mathbb{R}^{N \times L}$, defined as in 3.1.2, implicitly using matrix $\mathbf{Q}$ through the Householder vectors derived from $\mathbf{U}$, and compute output weight matrix $\boldsymbol{\beta} \in \mathbb{R}^{k \times L}$.

$$\mathbf{c} = \mathbf{Q}_1^T \cdot \mathbf{T} = \begin{bmatrix} \mathbf{T} \\ \mathbf{0} \end{bmatrix} - 2 \cdot \mathbf{u} \cdot \left(\mathbf{u}^T \cdot \begin{bmatrix} \mathbf{T} \\ \mathbf{0} \end{bmatrix} \right)$$

$$\boldsymbol{\beta} = \mathbf{R}^{-1} \cdot \mathbf{c}_{1:k}$$

A padding of zeros is added under the target matrix $\mathbf{T}$ to preserve the formula in 8 in paragraph 3.2.
The choice of using the Householder vectors instead of explicitly computing the matrix $\mathbf{Q}$ is explained in 5.1.

while stopping criterion not met **do**:

step 7: Increment $k = k + 1$

step 8: Add new column $\mathbf{w}_{new} \in \mathbb{R}^{M\times 1}$ of randomly chosen weights to $\mathbf{w}_k \in \mathbb{R}^{M\times k}$, as shown in 9, and recompute $\mathbf{H} \in \mathbb{R}^{N\times k}$ like in step 2, taking into account that k has been incremented

$$\mathbf{w}_k = \begin{bmatrix} \mathbf{w}_{k-1} & | & \mathbf{w}_{new} \end{bmatrix} \tag{9}$$

step 9: Like in step 3, the regularization coefficient is added to matrix $\mathbf{H}$, and matrix $\mathbf{E}$ is recomputed.

step 10: Compute Householder vector of the last column of matrix $\mathbf{E}$, $\mathbf{u} \in \mathbb{R}^{(N+1)\times 1}$, and $\mathbf{R}_{new} \in \mathbb{R}^{k\times 1}$, which is the new column for matrix $\mathbf{R}$, through the operations of the *qr_step* algorithm described in 3.3.1.

step 11: Update matrices $\mathbf{U} \in \mathbb{R}^{(N+1)\times k}$ by adding the new Householder vector column $\mathbf{u}$, and $\mathbf{R} \in \mathbb{R}^{k\times k}$ by inserting a row of $k-1$ zeros under matrix $\mathbf{R}$ and then appending column $\mathbf{R}_{new}$.

$$\mathbf{U} = \begin{bmatrix} \mathbf{U} & \mathbf{u} \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \quad \Bigg| \quad \mathbf{R}_{new} \tag{10}$$

Performing the update of $\mathbf{R}$ like in 10 preserves the fact that $\mathbf{R}$ is an upper triangular matrix.

step 12: Compute $\mathbf{c} \in \mathbb{R}^{(N+k-1)\times L}$ and $\boldsymbol{\beta} \in \mathbb{R}^{k\times L}$. A padding of zeros is added under $\mathbf{c}$ to preserve the formula in 8 in paragraph 3.2, since $\mathbf{c}$ is the result of the iterative multiplication between the target matrix $\mathbf{T}$ and the Householder vectors at each iteration k.

$$\mathbf{c} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}$$

$$\mathbf{c}_{k:N} = \mathbf{c}_{k:N} - 2 \cdot \mathbf{U}_{1:N+1,k} \cdot (\mathbf{U}^T_{1:N+1,k} \cdot \mathbf{c}_{k:N})$$

$$\boldsymbol{\beta} = \mathbf{R}^{-1} \cdot \mathbf{c}_{1:k}$$

end while

Steps *7* through *12* are repeated until the number of hidden neurons k reaches the maximum number of possible hidden neurons K_{max}, or the loss $\|\mathbf{Y} - \mathbf{T}\|_2$ computed during these steps is lower or equal than the established threshold accuracy ϵ, where $\mathbf{Y} = \mathbf{H} \cdot \boldsymbol{\beta}$ are predicted targets.

3.3.1 Qr_step Algorithm

In the following steps, the *qr_step* algorithm is described.

qr_step: Given a matrix $\mathbf{E} \in \mathbb{R}^{(N+k)\times k}$ to factor, a matrix $\mathbf{U} \in \mathbb{R}^{(N+1)\times k}$ to derive the Householder vectors of matrix $\mathbf{E}$'s columns and the number of actual hidden neurons k,

$\forall j = 1, ..., k-1$

step 1: Extract $\mathbf{u} \in \mathbb{R}^{(N+1+k-j)\times 1}$ to compute Householder vector of column j of matrix $\mathbf{E}$ and adds a column of $k-j$ zeros, according to the iteration on j

$$\mathbf{u} = \begin{bmatrix} \mathbf{U}_{1:N+1,j} \\ \mathbf{0} \end{bmatrix}$$

step 2: Multiply $\mathbf{E}$ with $\mathbf{u}$ to obtain the first $k-1$ elements of the new column of matrix $\mathbf{R}$ of the QR factorization

$$\mathbf{E}_{j:N+k,k} = \mathbf{E}_{j:N+k,k} - 2 \cdot \mathbf{u} \cdot (\mathbf{u}^T \cdot \mathbf{E}_{j:N+k,k}) \tag{11}$$

end loop

step 3: Compute $\mathbf{h} \in \mathbb{R}^{(N+1)\times 1}$ from which it is possible to compute the Householder vector of $\mathbf{E}_{k:N+k,k}$ and the value $s = \pm\|\mathbf{E}_{k:N+k,k}\|_2$, as stated in 3.1.1. The choice of this value is explained in 5.1.

step 4: Create the new column $\mathbf{R}_{new} \in \mathbb{R}^{k\times 1}$ of matrix $\mathbf{R}$

$$\mathbf{R}_{new} = \begin{bmatrix} \mathbf{E}_{1:k-1,k} \\ s \end{bmatrix}$$

This algorithm produces the new column $\mathbf{R}_{new} \in \mathbb{R}^{k \times 1}$ and $\mathbf{h} \in \mathbb{R}^{(N+1) \times 1}$ from which it is possible to derive the Householder vector of the last column of $\mathbf{E}$.

4 Standard momentum descent (heavy ball) approach

Momentum, also known as *Heavy Ball* method, is a variant of the gradient-based optimization techniques. It is designed to speed up learning of gradient based algorithms (for example, *Stochastic gradient descent*). The standard momentum technique uses the history of past gradients through the accumulation of a velocity vector in directions of reduction in the objective and continues to move in their direction [8].

Momentum is not a descent algorithm, there is no guarantee that the next iteration will be in a descent direction, but it may be necessary to get a smaller value of the function afterwards.

4.1 Standard momentum algorithm

Standard momentum algorithm: given a training set $(\mathbf{X}, \mathbf{T})$ with $\mathbf{X} = \{x_i[n]\} \in \mathbb{R}^{N\times I}$ denoting the input data and $\mathbf{T} = \{t_l[n]\} \in \mathbb{R}^{N\times L}$ the targets, a learning rate $\eta \in \mathbb{R}$, momentum hyperparameter $\alpha \in [0, 1) \subseteq \mathbb{R}$, initial parameters $\mathbf{w}$ and an initial velocity $\mathbf{v}$

while stopping criterion not met **do**:

step 1: At iteration k, sample a mini-batch of m examples of the training set with the corresponding targets

step 2: compute gradient estimate $\Delta\mathbf{w}_k$

step 3: compute velocity update

$$\mathbf{v}_{k+1} \leftarrow \alpha\mathbf{v}_k - \eta\Delta\mathbf{w}_k$$

step 4: apply update

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \mathbf{v}_{k+1}$$

end while

The variable $\mathbf{v}$ is called *velocity*, it represents the direction and speed at which the parameters move through parameter space, while the hyperparameter α determines how quickly the contributions of previous gradients exponentially decay.

Without computing the velocity vector explicitly, steps 3 and 4 can be rewritten as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta\Delta\mathbf{w}_k + \alpha(\mathbf{w}_k - \mathbf{w}_{k-1}) \tag{12}$$

4.2 Least squares with Standard momentum descent

This project's objective is using the standard momentum descent approach to solve the Least squares problem with L2-regularization, as stated in 3.2.

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left\| \begin{bmatrix} \mathbf{A} \\ \lambda\mathbf{Id} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \right\|_2^2$$

To be able to solve this problem with the standard momentum technique, it is needed to compute the gradient, as formulated in *step 2* of the algorithm 4.1. To ease this computation we can rewrite $min\|\mathbf{Ax} = \mathbf{b}\|$.

$$\begin{gathered} \min_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{Ax} - \mathbf{b}\|_2^2 = \\ \min_{\mathbf{x}\in\mathbb{R}^n} (\mathbf{Ax} - \mathbf{b})^T(\mathbf{Ax} - \mathbf{b}) = \\ \min_{\mathbf{x}\in\mathbb{R}^n} \mathbf{x}^T\mathbf{A}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{A}^T\mathbf{b} + \mathbf{b}^T\mathbf{b} = \\ \min_{\mathbf{x}\in\mathbb{R}^n} \mathbf{x}^T\mathbf{A}^T\mathbf{Ax} - 2\mathbf{b}^T\mathbf{Ax} + \mathbf{b}^T\mathbf{b} \end{gathered}$$

It follows that the gradient can be computed as in formula 13.

$$2\mathbf{A}^T\mathbf{Ax} - 2\mathbf{A}^T\mathbf{b} \tag{13}$$

As a consequence the Hessian is $2\mathbf{A}^T\mathbf{A}$, this implies that our problem is *strictly convex* because $\mathbf{A}^T\mathbf{A}$ is positive semidefinite with λ lower bound of the smallest eigenvalue, thus making the Hessian positive definite. This property guarantees that the minimum exists and is unique.

4.3 Analysis of Standard Momentum

As stated in theorem 9 of [7], Standard Momentum, which is an accelerated first-order method, has *linear convergence* when the objective function is twice continuously differentiable, strongly convex and has Lipschitz-continuous gradient.
In our specific case, Least squares satisfies the previous properties, which means the theorem is applicable and a linear convergence rate is expected.

In addition, in this case, it is possible to calculate the learning rate η and momentum coefficient α values through closed formulas, respectively 14 and 15.

$$\eta = \frac{4}{(\sqrt{L} + \sqrt{\tau})^2} \tag{14}$$

$$\alpha = \max\{|\ 1 - \sqrt{\eta\tau}\ |, |\ 1 - \sqrt{\eta L}\ |\}^2 \tag{15}$$

where $L < \infty$ is the Lipschitz constant and $\tau > 0$ is the modulus of convexity.
L and τ are, respectively, the largest and smallest eigenvalues, unfortunately the computation of the eigenvalues has cubic complexity so it is quite inefficient to calculate them. There are some estimates which can be used to bound these values, in particular, looking at the Hessian of section 4.2, τ can be lower bounded by λ, instead L has as upper bound the Frobenius norm.

In this project, these estimates are used to compute η and α as in formulas (14, 15), but they were also employed to perform a grid search to choose the values using the following bounds $\frac{1}{L} \leq \eta \leq \frac{4}{L}, \alpha \leq 1$ derived form the same formulas.

4.4 Algorithm ELM with Standard Momentum

Algorithm ELM with Standard Momentum: Given a training set $(\mathbf{X}, \mathbf{T})$ with $\mathbf{X} = \{x_i[n]\} \in \mathbb{R}^{N \times I}$ denoting the input data and $\mathbf{T} = \{t_l[n]\} \in \mathbb{R}^{N \times L}$ the targets, with activation function $\mathrm{g}(\cdot)$, and maximum number of iterations J_{max}.
Also, given $\epsilon \in \mathbb{R}$ threshold for accuracy of the results, $\alpha \in \mathbb{R}$ momentum coefficient and $\lambda \in \mathbb{R}$ regularization coefficient,

step 1: Assign arbitrary input weights $\mathbf{w}_j \in \mathbb{R}^{M \times K}$ and bias $b_j = w_{0k}$, with $k = 1$, where $M = I + 1$ because of the inclusion of the bias in the weights.

step 2: Calculate the hidden layer output matrix $\mathbf{H} \in \mathbb{R}^{N \times K}$ as in 1, and initialize randomly output weight matrix $\boldsymbol{\beta} \in \mathbb{R}^{K \times L}$.

step 3: Add regularization coefficient to matrix $\mathbf{H}$, where $\mathbf{Id} \in \mathbb{R}^{K \times K}$ identity matrix, to generate matrix $\mathbf{E} \in \mathbb{R}^{(N+K) \times K}$. Matrix $\mathbf{E}$ is the new matrix to factor.

$$\mathbf{E} = \begin{bmatrix} \mathbf{H} \\ \lambda \mathbf{Id} \end{bmatrix}$$

step 4: both matrices $\mathbf{E}$ and $\mathbf{T}$ are multiplied on the left by $\mathbf{E}^T$, which will still solve the original Least squares problem, but with the new formulation $\mathbf{E}^T\mathbf{E}\boldsymbol{\beta} = \mathbf{E}^T\mathbf{T}$, maintaining the properties of symmetricity and positive definiteness.

while stopping criterion not met **do**:

step 5: Computation of the gradient similarly as in 4.2, without the constants and

taking into consideration that $\mathbf{E}$ and $\mathbf{T}$ were already multiplied by $\mathbf{E}^T$.

$$\Delta\boldsymbol{\beta}_j = \mathbf{E}\boldsymbol{\beta} - \mathbf{T}$$

step 6: Update output weights $\boldsymbol{\beta}$ as in 12.

$$\boldsymbol{\beta}_{j+1} = \boldsymbol{\beta}_j - \eta\Delta\boldsymbol{\beta}_j + \alpha(\boldsymbol{\beta}_j - \boldsymbol{\beta}_{j-1})$$

Steps *5* and *6* are repeated until the maximum number of iterations J_{max}, or the loss $\|\mathbf{Y} - \mathbf{T}\|_2$ computed during these steps is lower or equal than the established threshold accuracy ϵ, where $\mathbf{Y} = \mathbf{E} \cdot \boldsymbol{\beta}$ are predicted targets.

5 Implementation

For this project, an *Incremental Extreme Learning Machine* (I-ELM) based on *QR factorization* and an *Extreme Learning Machine* using the *standard momentum approach*, both with *L2-regularization*, were implemented using the **MATLAB** programming language. In this section are documented the implementation choices of both of the algorithms used in this project, and the files which contain the main functions are listed.

5.1 QRI-ELM

To train QRI-ELM two different functions were created: for experimental purposes one implementation takes in input a pre-computed matrix from which the weights are extracted (column by column), the other instead generates a new column of weights at run-time.

For the latter, the creation of the network weights uses the `init.m` function, which only takes as arguments the number of input features and output features, this is a consequence of the fact that in Incremental-ELM the hidden layer is initialized with a single neuron and new connections are added at each step of the training process.

The `train.m` function of I-ELM implements the algorithm shown in 3.3, it takes in input the weights, the Training Set $\mathbf{X} \in \mathbb{R}^{N \times I}$ (N being the number of samples and I the number of features), the targets $\mathbf{T} \in \mathbb{R}^{N \times L}$ with L output features, the threshold for the loss, the maximum number of neurons, the hidden layer activation function and the regularization coefficient λ.
During the execution of this function neurons are added incrementally to the hidden layer until the computed gap for the current step is not below the set threshold or until the maximum number of neurons for the hidden layer is reached. To reduce the computational complexity of this process a series of optimizations have been applied to different operations:

1. At each iteration, the QR factorization is only computed for the newly added column of the hidden layer output matrix $\mathbf{H}$ through the use of the `qr_step.m` function which computes the new column for the matrix $\mathbf{R}$ and the u-vector to compute the Householder vector of the last column of $\mathbf{H}$, which are then concatenated to the respective matrices $\mathbf{R}$ and $\mathbf{U}$.
   The new column $\mathbf{R}_{new} \in \mathbb{R}^{k \times 1}$ of $\mathbf{R}$ is computed such that the first $k-1$ entries are the elements resulting from the iterative multiplication of the columns of the matrix to factor and the respective Householder vectors, and the k-th element is $s = \pm\|\mathbf{x}\|_2$, where $\mathbf{x}$ is the last column of the matrix to factor.

2. To reduce the space complexity, the matrix $\mathbf{Q}$ is never stored explicitly, in particular

only the u-vectors to derive the Householder vectors are stored in a matrix, **U**. The `qr_step.m` function multiplies iteratively the vectors in this matrix with the new column of **H** to create the correct input for the computation of the new Householder vector.

3. For every multiplication in which the **Q** matrix would be involved, formula 11 has instead been applied. For example, in a simplified version to measure the complexity of this operation, given a column vector $\mathbf{a} \in \mathbb{R}^{N\times 1}$ and the vector $\mathbf{u} \in \mathbb{R}^{N\times 1}$ (to compute the Householder vector), with compatible dimensions, we have:

$$\mathbf{a} = \mathbf{a} - 2 \cdot \mathbf{u} \cdot (\mathbf{u}^T \cdot \mathbf{a}) \tag{16}$$

   Without the use of parenthesis, operation 16 would first have a vector-vector product, which generates a square matrix, and then a matrix-vector product. To optimize this, the use of parenthesization allows computing a vector-vector multiplication, which instead results in a scalar that is then multiplied by a vector. The first case would have $O(N^2)$ complexity, which is then reduced to $O(N)$ complexity in the second, optimized, case.

4. To optimize the time and space complexities the factorization computed is a thin QR factorization.

Furthermore, the Householder vectors are computed through the use of a specific function `householder_vector.m`, which outputs the vector and the value $s = \pm\|\mathbf{x}\|_2$, this value is chosen to avoid incurring in numerical problems. Specifically,

$$s = \begin{cases} -\|\mathbf{x}\|_2 & if \;\; \mathbf{x}_1 \geq 0 \\ +\|\mathbf{x}\|_2 & otherwise \end{cases}$$

where $\mathbf{x}_1$ is the first element of vector $\mathbf{x}$ of which we are computing the Householder vector.

5.2 ELM with Standard Momentum

The network weights are initialized in the `init.m` function which takes as parameters the number of input features, the number of hidden and output neurons and an optional argument which is the seed (integer) to manage the randomization of the weights and to be able to reproduce experiments.

The `train.m` function implements the algorithm shown in 4.4. Similarly to the previous implementation, it takes in input the weights, the Training Set $\mathbf{X} \in \mathbb{R}^{N\times I}$ (N being

the number of samples and I the number of features), the targets $\mathbf{T} \in \mathbb{R}^{N \times L}$ with L output features, the threshold for the gap, the hidden layer activation function and the regularization coefficient λ, but also the maximum number of iterations, the learning rate η and the momentum coefficient α.

6 Experiments

In this section the phase regarding the experiments and their results is described.

First, the experiments on QRI-ELM were conducted and then the ones on ELM with Standard Momentum. To be able to confront the two algorithms, it was necessary to fix the starting LS problems. In particular, this was done by choosing as activation function the *sigmoid* and four different regularization coefficient values to study how the solutions vary. Also, the hidden layer weights of both networks must be the same to be able to produce the same hidden layer output matrix $\mathbf{H}$. As a matter of fact, for ELM with Standard Momentum the initial hidden weights used were the same as the ones for the configurations of QRI-ELM.

As stopping criterion, other than the maximum number of neurons (for QRI-ELM) or iterations (for ELM with Standard Momentum), a threshold value was set.

To measure the results of both algorithms the following relative gaps were used 17, 18.

$$gap\_sol = \frac{\|\boldsymbol{\beta} - \boldsymbol{\beta}^*\|}{\|\boldsymbol{\beta}^*\|} \tag{17}$$

where $\boldsymbol{\beta}^*$ is computed as $\mathbf{H}^{-1}\mathbf{T}$.

$$gap\_pred = \frac{\|\mathbf{H}\boldsymbol{\beta} - \mathbf{T}\|}{\|\mathbf{T}\|} = \frac{\|\mathbf{Y} - \mathbf{T}\|}{\|\mathbf{T}\|} \tag{18}$$

Specifically for ELM with Standard Momentum, additional tests were performed as stated in 4.3.

6.1 Dataset

The dataset employed for these experiments is the **ML-CUP** of the Machine Learning course of the University of Pisa from the academic year 2021/2022. This dataset is composed of 10 columns of input features of continuous numerical attributes. Also the target has 2 output features which are continuous values, meaning the task to solve is a *regression task*.

The whole dataset is composed of *1477* examples.

6.2 Property of our LS problem

To study the expected values of the aforementioned metrics (17, 18) it is necessary to analyze the starting LS problem for each configuration of the regularization coefficients λ.

Given our input dataset described in 6.1, the Extreme Learning Machine will compute the hidden layer output matrix $\mathbf{H}$ as described in 1. In particular the hidden layer weight matrix which is used to calculate $\mathbf{H}$ was generated and fixed for all the experiments; specifically, for QRI-ELM given the fact that it builds its matrices incrementally, at each step of the algorithm sub-matrices composed of columns of this weight matrix were selected until one of the stopping criteria is met.

To matrix $\mathbf{H}$ will be added the regularization as stated in 8 producing matrix

$$\mathbf{H}_{reg} = \begin{bmatrix} \mathbf{H} \\ \lambda \mathbf{Id} \end{bmatrix}$$

and the goal is to find the solution $\boldsymbol{\beta}$ which are the output weights of the network.

Initially, the condition number of matrix $\mathbf{H}$ was analyzed and it was discovered that $\mathcal{K}(\mathbf{H}) \approx 1e3$ meaning that the matrix has a high-condition number but is not ill-conditioned. Adding the regularization and studying the condition number of matrix $\mathbf{H}_{reg}$, it turned out that this is inversely proportional to regularization, the more it is reduced the greater the conditioning becomes, as shown in the table 1.

λ	$\#Neurons$	$\mathcal{K}(\mathbf{H}_{reg})$
1e1	150	2.37e1
1	150	2.36e2
1e-2	150	2.96e+03
1e-4	150	2.98e+03

Table 1: Condition numbers studied as the regularization coefficient varies, where matrix $\mathbf{H}_{reg}$ was the one used for both QRI-ELM and ELM with Standard Momentum.

6.3 QRI-ELM Experiments

As for the Incremental Extreme Learning Machine, its parameters are: *k_max* maximum number of hidden neurons, *activation function*, the regularization coefficient λ. As stated in the introduction of this section, the activation function is set as *sigmoid*. The maximum number of neurons is set to 150. The other stopping criterion is based on *gap_pred* with a threshold of 1e-10.

The final results of the experiment are shown in Table 2.

λ	*#Neurons*	*Gap_sol*	*Gap_pred*	$\mathcal{K}_{rel}(LS, \mathbf{T}) \cdot \frac{\|r_1\|}{\|\mathbf{T}\|}$
1e1	150	4.32e-15	1.01e-1	7.98e-15
1	150	5.08e-15	6.53e-2	6.86e-14
1e-2	150	3.96e-14	5.57e-2	1.41e-12
1e-4	150	5.15e-14	5.57e-2	1.29e-12

Table 2: Results of QRI-ELM, where *#Neurons* is the number of the neurons reached by the network given the upper bound *k_max* and threshold for the *gap_pred*.

The table above also shows the expected bound under which we expect to obtain the results of the difference between the calculated solution and the optimal solution as shown in formula 19.

$$\frac{\|\boldsymbol{\beta} - \boldsymbol{\beta}^*\|}{\|\boldsymbol{\beta}^*\|} \leq \mathcal{K}_{rel}(LS, \mathbf{T}) \cdot \frac{\|r_1\|}{\|\mathbf{T}\|} \tag{19}$$

where

$$\mathcal{K}_{rel}(LS, \mathbf{T}) = \mathcal{K}(\mathbf{H}_{reg}) \cdot \frac{\|\mathbf{T}\|}{\|\mathbf{H}_{reg}\boldsymbol{\beta}\|}$$

and *r1* is the residual calculated as follow:

$$r_1 = \mathbf{Q}_1^T(\mathbf{H}_{reg}\boldsymbol{\beta} - \mathbf{T})$$

In accordance with theory, the gap to the optimal solution *gap_sol* is always smaller than the calculated upper bound. Even though the bound increases as the regularization decreases, *gap_sol* stays more or less in the same order of magnitude without going too near to the bound.
Also, each configuration of the QRI-ELM network reached the maximum number of neurons in less than 15 seconds of training.

Another part of the experiments focused on studying how accurately the QRI algorithm factorizes the matrix $\mathbf{H}_{reg}$, in particular Table 3 shows that the relative gap between $\mathbf{H}_{reg}$ and the obtained factorization is always near machine precision.

λ	$\frac{\|\mathbf{QR} - \mathbf{H}_{reg}\|}{\|\mathbf{H}_{reg}\|}$
1e1	5.32e-16
1	1.54e-16
1e-2	3.70e-16
1e-4	3.29e-16

Table 3: Accuracy of QR factorization with respect to the $\mathbf{H}_{reg}$ matrix

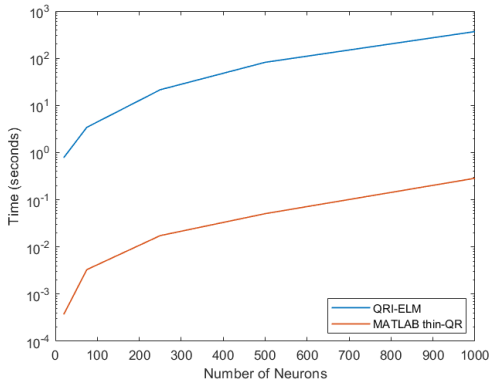
A comparison with the *MATLAB thin-QR factorization* was carried out to evaluate if there are any significant changes in performance between an incremental thin-QR and one which acts on the complete matrix. In terms of *gap_sol* and *gap_pred* the results differ in the order of *e-15*.

In order to evaluate the **scalability** of the algorithm with QRI-ELM, different configurations of number of neurons were tested, thus varying the size of the starting matrix, in order to measure the execution time. The upper bound of number of neurons was always reached.

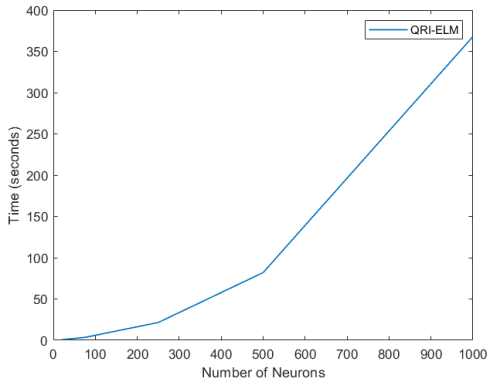
λ	*#Neurons*	*Gap_sol*	*Gap_pred*	*Time* (seconds)
1e-2	20	1.05e-15	8.36e-2	0.78
	75	4.12e-15	6.00e-2	3.43
	250	1.08e-14	5.10e-2	21.51
	500	3.00e-14	4.46e-2	82.10
	1000	8.68e-14	3.24e-2	386.35

Table 4: Scalability of QRI-ELM

As shown in Table 4, the execution time increases as does the size of the matrix to be factorized. In particular, only a value of a regularization coefficient equal to *1e-2* is reported because this does not affect the time of execution.



(a) Comparison of QRI-ELM and MATLAB thin-QR, with y-axis in log-scale



(b) QRI-ELM time of execution with respect to number of neurons

Figure 1: Experiments on execution time with respect to the number of neurons, with $\lambda = 1e-2$

Finally, the influence of the number of neurons on the running time of the algorithm is depicted in Figure 1a where QRI-ELM is compared with MATLAB's thin-QR using the

same regularization coefficient, $\lambda = 1e-2$, and the same number of neurons as the ones in Table 4. In particular, even though our algorithm results in higher execution times, it behaves similarly to MATLAB's thin-QR. This behavior is coherent with theoretical complexity (for a $m \times n$ matrix with $m \gg n$, $O(mn^2)$), as shown in Figure 1b.

6.4 ELM with Standard Momentum Experiments

This algorithm presents the following parameters: the *activation function*, the *number of hidden neurons*, the regularization coefficient λ, the learning rate η, the momentum coefficient α, j_max maximum number of iterations.

In addition to the previously fixed values, i.e. the activation function ("*sigmoid*") and number of neurons (150), also the maximum number of iterations j_max was set to *60'000*. The additional stopping criterion is based on the norm of the gradient with a threshold value of 1e-10.

For the *learning rate* and *momentum* parameters different analyses were performed considering the formulas 14, 15. The analyses were carried out using the approximations of L and τ described in 4.3.

Afterwards, they were compared with a model trained with the same parameters but with the *optimal* values of learning rate and momentum (i.e. by computing explicitly the minimum and maximum eigenvalues without any approximations) to study how much the performance would improve.

Additional test were performed making use of the bounds given by the theory to verify how much the results would change without using specific formulas. For the learning rate, the tested values were the lower bound ($\frac{1}{L}$), the upper bound ($\frac{4}{L}$) and a variable number of random values in this interval. Instead, for the momentum parameter, different fixed values which respect the upper bound (≤ 1) were tested.

In Table 5 are reported the results of experiments using the values computed through the formulas from theory.

In Table 6 performances on values of the learning rate and momentum generated within the bounds given by the theory are depicted, taking into consideration the best results for each configuration of regularization coefficient.

Finally, in Table 7 results with optimal values of learning rate and momentum are shown.

λ	η	α	*#Iter*	*Gap_sol*	*Gap_pred*
1e1	6.87e-5	0.95	845	2.20e-9	8.16e-2
1	7.01e-5	0.98	2491	6.67e-8	6.09e-1
1e-2	7.07e-5	0.99	31070	4.20e-5	5.57e-2
1e-4	7.07e-5	0.99	60000	1.56e-1	7.77e-2

Table 5: Results of ELM with Standard Momentum with formulas of η and α using approximations.

λ	η	α	*#Iter*	*Gap_sol*	*Gap_pred*
1e1	1.76e-5	0.95	866	6.07e-9	8.16e-2
1	6.11e-05	0.8	43302	6.79e-7	6.08e-2
1e-2	6.94e-05	0.95	60000	3.62e-1	5.57e-2
1e-4	6.83e-05	0.95	60000	3.39e-1	5.57e-2

Table 6: Results of ELM with Standard Momentum with values generated in the bounds.

λ	η	α	*#Iter*	*Gap_sol*	*Gap_pred*
1e1	6.57e-5	0.84	352	6e-11	8.16e-2
1	7.09e-5	0.98	3766	1.71e-11	6.10e-2
1e-2	7.14e-5	0.99	51094	2.71e-11	5.57e-2
1e-4	7.14e-5	0.99	51611	2.69e-11	5.57e-2

Table 7: Results of ELM with Standard Momentum with optimal values of η and α.

Taking into consideration the values of the learning rate η and momentum coefficient α, it is possible to notice that these values are similar between formulas using the approximations of L and τ and the ones with the optimal values, but with a noticeable difference in the *gap_sol* results. Instead, as for the random values within the bounds which gave the best results, the learning rate and momentum are different from the others.

Analyzing the maximum number of iterations reached, with the decrease of the regularization value, it can be seen that it increases significantly up to reaching the upper bound of *60'000* iterations. In some cases, for example with regularization coefficient values of *1e1* and *1e-4* using the optimal values, the algorithm converges in a number of iterations smaller than when using the approximations.

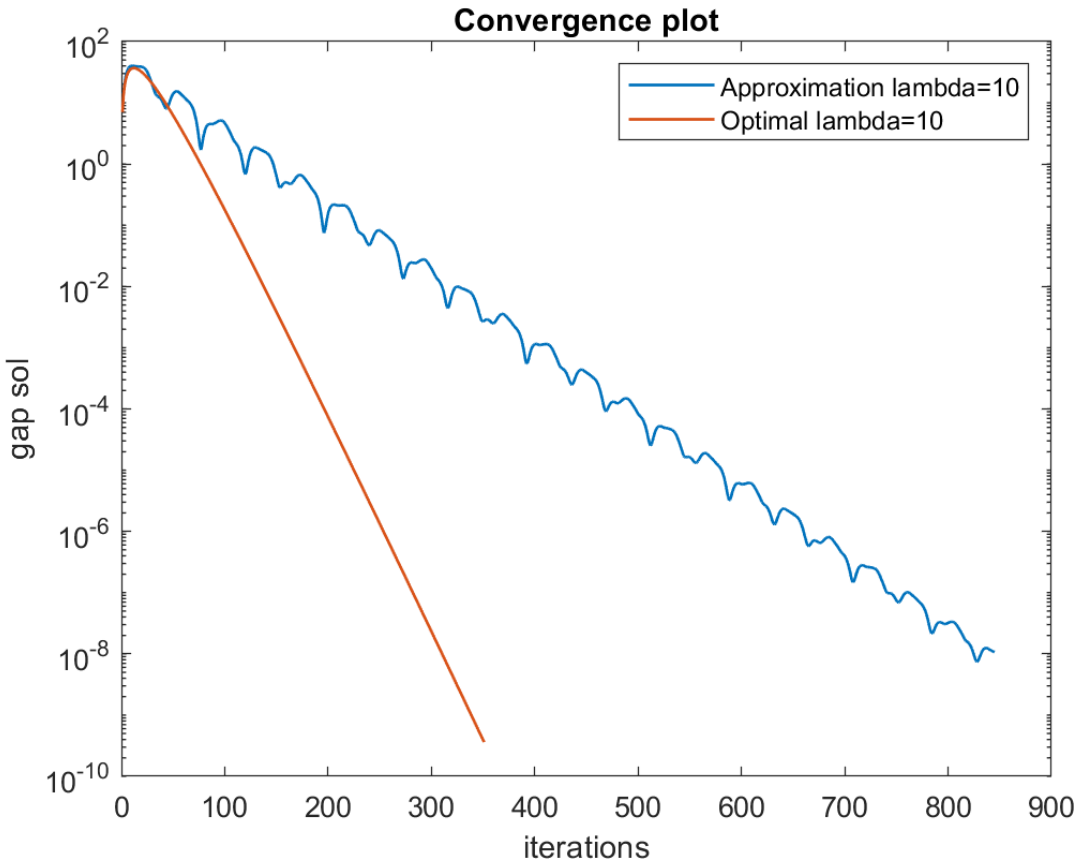


Figure 2: Converge of Standard Momentum using optimal and approximation values for regularization coefficient of 1e1, with y-axis in log-scale.

As it is possible to see in Figure 2, using the optimal values a faster and smoother convergence is obtained, whereas with the approximation a worse *gap_sol* is achieved after a lot of zig-zagging.

Finally, in terms of *gap_sol* using the optimal values considerably better results are achieved. For the other approaches, with the decrease of the regularization value, the *gap_sol* worsens substantially.

For all the approaches, the *gap_pred* value is more or less the same with only slightly bigger values when using the approximated values of L and τ.

As for the execution time, it is proportional to the number of iterations, going from a minimum of less than 1 second up to $\approx$ 2.5 seconds when the maximum number of iterations is reached.

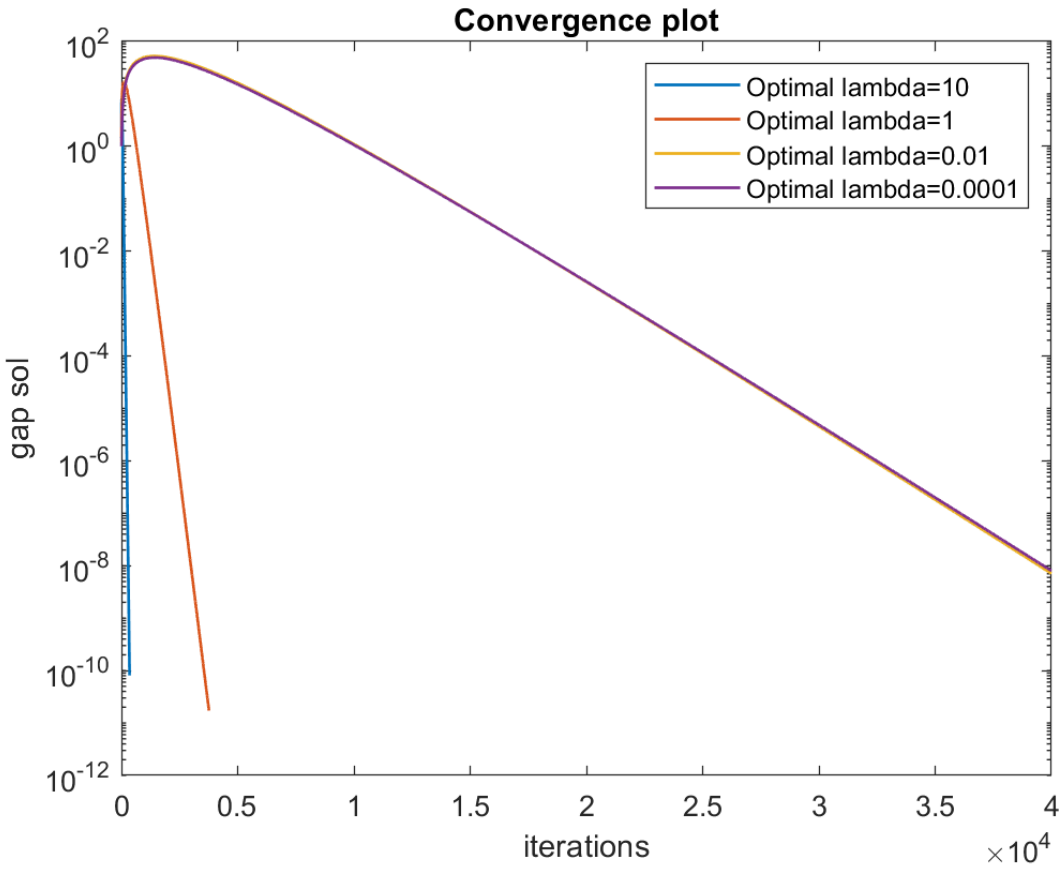


Figure 3: Converge of Standard Momentum with different regularization coefficients using optimal values, with y-axis in log-scale.

In Figure 3 is represented how *gap_sol* varies with different values of the regularization coefficient λ when using the optimal values of the learning rate and momentum coefficient. As per theory, when the λ value increases the convergence is significantly faster, as shown in Table 7, going from just a couple hundred iterations when $\lambda = 1e1$ to over $50'000$ when $\lambda = 1e-4$.

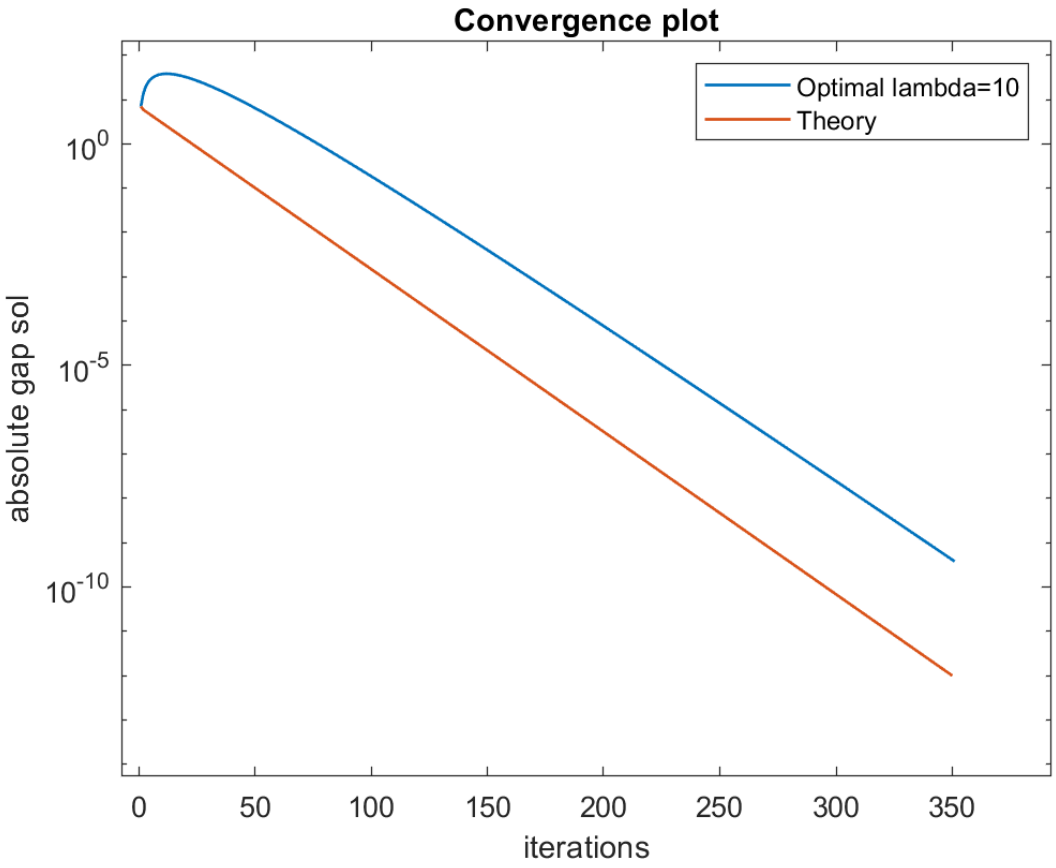


Figure 4: Converge of Standard Momentum with optimal values and generic linear convergence, with y-axis in log-scale.

In Figure 4 is depicted the convergence of Standard Momentum using optimal values with regularization coefficient $\lambda = 10$, where *absolute_gap_sol* is computed as follows:

$$absolute\_gap\_sol(i) = \|\boldsymbol{\beta}^i - \boldsymbol{\beta}^*\| \tag{20}$$

where $\boldsymbol{\beta}^*$ is computed as $\mathbf{H}^{-1}\mathbf{T}$.

Instead, the theorical convergence is computed as:

$$r^i \cdot absolute\_gap\_sol(0) \tag{21}$$

where i is the number of the current iteration and r is the convergence rate, calculated as:

$$r = \frac{\sqrt{L} - \sqrt{\tau}}{\sqrt{L} + \sqrt{\tau}} \tag{22}$$

This (21) convergence is the typical linear convergence. According to Gelfand's formula [6], where $\rho(C)$ is the spectral radius:

$$\rho(C) = lim_{i\to\infty}\|C^i\|^{1/i} \tag{23}$$

$$C^i = \begin{bmatrix} (1+\alpha)\mathbf{I} - \eta\nabla f^2(\boldsymbol{\beta}^*) & -\alpha\mathbf{I} \\ \mathbf{I} & 0 \end{bmatrix}$$

which means that $\forall\epsilon > 0\ \exists h\ s.t.\ \|C^i\| \leq (\rho(C)+\epsilon)^i\ \forall i \geq h$, so after h iterations the method should converge, resulting in a *quasi-linear* convergence rate. In our case, as shown in Figure 4, Standard Momentum grows in the first iterations and then assumes a convergence similar to the generic linear convergence, this behavior is compliant to the aforementioned theory.

Finally, **scalability** was also evaluated for ELM with Standard Momentum by distinguishing different numbers of neurons for each regularization coefficient. From the results shown in Table 8, it can be seen that the execution time is influenced by the regularization coefficient value. In two configurations, with small values of λ and 1000 neurons, the stopping criterion is met and the quality of these solutions both in terms of *gap_sol* and *gap_pred* is greatly inferior with respect to the others.

In addition, configurations with $\lambda = 1e-2$ and $\lambda = 1e-4$ behave similarly, as expected from the previous experiment shown in Figure 3.

λ	*#Neurons*	*#Iter*	*Gap_sol*	*Gap_pred*	*Time* (seconds)
1e1	20	119	5.62e-11	1.07e-1	0.06
	75	244	6.21e-11	7.61e-2	0.09
	250	466	6.72e-11	6.69e-2	0.21
	500	658	1.07e-10	6.10e-2	0.48
	1000	970	1.55e-10	5.75e-2	1.41
1	20	587	5.49e-11	8.37e-2	0.07
	75	2119	3.76e-11	6.04e-2	0.24
	250	4791	2.73e-11	5.24e-2	0.46
	500	7062	1.35e-11	4.93e-2	1.08
	1000	10462	1.00e.11	4.44e-2	6.68
1e-2	20	667	5.12e-11	8.36-2	0.07
	75	3710	3.28e-11	6.00e-2	0.23
	250	17656	8.72e-12	5.10e-2	1.35
	500	56921	1.53e-11	4.46e-2	5.21
	1000	60000	1.13	154.58	33.70
1e-4	20	667	5.12e-11	8.36e-2	0.07
	75	3717	2.87e-11	6.00e-2	0.25
	250	17529	1.37e-11	5.10e-2	1.31
	500	57386	1.56e-11	4.46e-2	5.32
	1000	60000	1.57	225.50	33.57

Table 8: Scalability of ELM with Standard Momentum

7 Conclusions

Evaluating the results obtained from both approaches, it is possible to define certain characteristics found in the use of **QRI-ELM** and **ELM with Standard Momentum**, which highlight the advantages and disadvantages of both approaches.

Taking into account the **quality of the solution** produced at the end of the execution of both algorithms, which is computed in terms of *gap_sol*, QRI-ELM outperforms ELM with Standard Momentum because it almost reaches machine precision. Instead ELM, even when using the optimal values of learning rate and momentum, reaches at best a *gap_sol* of ≈1e-11.

In general, the ELM approach with the optimal values would not be trivial on real-world problems because of its computational complexity as previously explained. This means that it is important to consider the results obtained with the other values because they are more likely to be employed for more complex problems.

Something interesting that was found analyzing these results is the **impact of the regularization coefficient** on the performance of ELM with Standard Momentum. In particular, as the λ value decreases, the solution worsens considerably, as does the number of iterations, and thus the **time** required for the algorithm to converge. For greater regularization coefficient values, the time spent to find the solution is less than half a second with 150 neurons, instead for smaller values a time of up to ≈ 2.5 seconds is achieved.

QRI-ELM was not instead influenced by the changes in the λ value, as a matter of fact the time of convergence stays constant at around 15 seconds for 150 neurons. This shows that ELM is faster, but less precise.

Moreover, in these experiments' case, the main advantage of QRI-ELM to automatically find the **number of neurons** was not useful because the established upper bound was always met. Instead, for ELM with Standard Momentum, due to the experimental setting, it was not possible to determine if our problem could be solved by a different number of neurons. Then, in a real-world case it is necessary to spend time to find the optimal number of neurons which represents a good trade-off between the complexity of the model and performance.

In terms of **scalability**, QRI-ELM suffers the most from the increase in columns, as expected from theorical complexity, while the accuracy only loses a little bit of precision in terms of *gap_sol* but still gives a really good result, kind of near machine precision. Instead, the scalability of ELM is greatly influenced by the regularization coefficient value: for larger values a small number of iterations (and thus time) is needed for convergence, on the contrary for smaller λ values a greater number of iterations is always needed, up

to meeting the stopping criterion of maximum number of iterations. Specifically, when a small regularization coefficient is used along with a huge number of neurons the produced solutions have a quality that is not acceptable, as shown in Table 8, in these cases it is preferable to use the QRI-ELM approach. In general, ELM with Standard Momentum is faster, but with a little bit less accurate solutions (≈e-11 for ELM and ≈e-15 for QRI-ELM). The ELM with Standard Momentum approach results in a better trade-off between the accuracy and execution time for most configurations.

For what concerns the *gap_pred*, there is not much difference among the two algorithms, apart from the aforementioned case.

In conclusion, one of the main goals of this work was to explore the two approaches in the convex optimization case. In particular, given the results obtained from the experimentation, it was possible to say that the ELM approach with Standard Momentum, having more parameters to manage implies either a search for the best values, which can become difficult, or the use of approximations for the formulas, especially when optimal values cannot be derived. In contrast, QRI-ELM overcomes this problem and has shown that the quality of the obtained factorization is not affected by the incremental computations.

References

[1] Ben-Israel A. Generalized inverses of matrices and their applications. *Extremal Methods and Systems Analysis*, 174:154–186, 1980.

[2] Guangbin Huang, Ming-Bin Li, Lei Chen, and Chee Kheong Siew. Incremental extreme learning machine with fully complex hidden nodes. *Neurocomputing*, 71:576–583, 2008.

[3] Guangbin Huang, Qin-Yu Zhu, and Chee Kheong Siew. Extreme learning machine: a new learning scheme of feedforward neural networks. *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, 2:985–990 vol.2, 2004.

[4] Guangbin Huang, Qin-Yu Zhu, and Chee Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70:489–501, 2006.

[5] J.M. Ortega. *Matrix Theory*. Plenum Press, New York, London, 1987.

[6] Boris Polyak. *Introduction to Optimization*. 07 2020.

[7] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[8] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[9] L.N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[10] Yibin Ye and Yang Qin. Qr factorization based incremental extreme learning machine with growth of hidden nodes. *Pattern Recognition Letters*, 65:177–183, 11 2015.