UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Computational Mathematics

ML project 19

Professori:

Prof: Antonio Frangioni

Prof: Federico Poloni

Gruppo 13:

Filippo Morelli
Saul Urso

ANNO ACCADEMICO 2023/2024

Contents

Chapter 1

Problem definition

Extreme Learning Machines (ELMs) represent a significant departure from traditional neural network training methodologies. Unlike conventional feedforward neural networks, which rely on backpropagation through multiple network layers and often face challenges such as the vanishing gradient problem in deep architectures, ELMs simplify the training process. In ELMs, only the output layer is trainable, while the hidden layer parameters are randomly assigned and fixed. This unique approach eliminates the issues associated with deep network training, offering a more efficient and effective alternative.

An ELM with an hidden layer size of h is defined as :

$$f(x) = \sigma(x^T W_{in}) W^*_{out} \tag{1.1}$$

where:

- $x \in \mathbb{R}^n$ is a sample of n features;
- $W_{in} \in \mathbb{R}^{n \times h}$ is a fixed random matrix;
- $\sigma(\cdot)$ is an elementwise non-linear activation function;
- $W^*_{out} \in \mathbb{R}^{h \times m}$ is the output weight matrix, chosen by solving a linear least-squares problem with L_2 regularization;

In order to obtain W^*_{out} we first need two matrices:

- a **data matrix** $X \in \mathbb{R}^{L \times n}$ made of L samples of n features (each sample is a "row");
- a **target matrix** $Y \in \mathbb{R}^{L \times m}$ made of L targets of size m (each target is a "row").

In the general case, a least square problem is of the following form

Definition 1.0.1 (**Generic least-square problem**). *Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $y \in \mathbb{R}^m$ we want to find $w \in \mathbb{R}^n$ that solves:*

$$\min_{w \in \mathbb{R}^n} \|Aw - y\|_2^2$$

In our case however we do not have a vector w but a matrix $W^*_{out} \in \mathbb{R}^{h \times m}$, so the difference inside the norm in our case becomes:

$$\sigma(XW_{in})W^*_{out} - Y = AW^*_{out} - Y \qquad \text{with } A = \sigma(XW_{in})$$

Now let us look at the structure of this difference matrix. Let us call $w^*_{:i}$ and $y_{:i}$ the columns of W^*_{out} and Y respectively. Thus the difference matrix above can be written as:

$$AW^*_{out} - Y = \left(Aw^*_{:1} - y_{:1} \;\middle|\; Aw^*_{:2} - y_{:2} \;\middle|\; \dots \;\middle|\; Aw^*_{:m} - y_{:m}\right)$$

We can thus define the least-square problem for the ELM training as follows.

Definition 1.0.2 (**L2-regularized ELM least-squares problem**). *Given the matrices $A \in \mathbb{R}^{L\times h}$ and $Y \in \mathbb{R}^{L\times m}$, we want to find $w^*_{:1}, w^*_{:2}, \dots, w^*_{:m} \in \mathbb{R}^h$ such that:*

$$w^*_{:1}, w^*_{:2}, \dots, w^*_{:m} = \underset{w_{:1},w_{:2},\dots,w_{:m}\in\mathbb{R}^h}{\arg\min} \sum_{i=1}^{m} \|Aw_{:i} - y_{:i}\|_2^2 + \alpha\|w_{:i}\|_2^2 \quad \textit{with } \alpha > 0$$

or alternatively:

$$W^*_{out} = \underset{W_{out}\in\mathbb{R}^{h\times m}}{\arg\min} \|AW_{out} - Y\|_F^2 + \alpha\|W_{out}\|_F^2 \qquad \textit{with } \alpha > 0$$

Where $\|\cdot\|_F$ is the Frobenius norm and $\alpha > 0$ is the regularization hyperparameter, which controls the trade-off between model complexity and generalization capability.

The L2-regularized least-squares problem can also be reformulated as a standard least-squares problem:

$$\begin{aligned}
W^*_{out} &= \underset{W_{out}\in\mathbb{R}^{h\times m}}{\arg\min} \|AW_{out} - Y\|_F^2 + \alpha\|W_{out}\|_F^2 \\
&= \underset{W_{out}\in\mathbb{R}^{h\times m}}{\arg\min} \frac{1}{2}\|AW_{out} - Y\|_F^2 + \frac{1}{2}\left\|\sqrt{\alpha}W_{out}\right\|_F^2 \\
&= \underset{W_{out}\in\mathbb{R}^{h\times m}}{\arg\min} \frac{1}{2}\left\|\begin{pmatrix} AW_{out} - Y \\ \sqrt{\alpha}W_{out} - 0 \end{pmatrix}\right\|_F^2 \\
&= \underset{W_{out}\in\mathbb{R}^{h\times m}}{\arg\min} \frac{1}{2}\left\|\begin{pmatrix} A \\ \sqrt{\alpha}I \end{pmatrix}W_{out} - \begin{pmatrix} Y \\ 0 \end{pmatrix}\right\|_F^2 \\
&= \underset{W_{out}\in\mathbb{R}^{h\times m}}{\arg\min} \frac{1}{2}\|BW_{out} - Z\|_F^2
\end{aligned} \tag{1.2}$$

Notice that for $\alpha > 0$ the matrix:

$$B = \begin{pmatrix} A \\ \sqrt{\alpha}I \end{pmatrix} \in \mathbb{R}^{(L+h)\times h}$$

has always full rank h. Since B has full column rank we can state the following proposition.

Proposition 1.0.1 (**Positive definiteness of M^TM**). *Let $M \in \mathbb{R}^{n\times m}$ be a full column rank matrix. M has full column rank iff. M^TM is symmetric and positive definite.*

Proof. We first prove M^TM is symmetric:

$$(M^TM)^T = M^T(M^T)^T = M^TM$$

To prove M^TM is positive definite we first prove M^TM is positive semidefinite. For M^TM to be positive semidefinite we need:

$$x^T(M^TM)x \geq 0 \quad \forall x \in \mathbb{R}^m$$

but notice that:

$$x^T(M^TM)x = (Mx)^TMx = \|Mx\|_2^2 \geq 0$$

Thus M^TM is positive semidefinite. Since M also has full-column rank notice that:

$$\text{M has full-column rank} \iff Mz \neq 0 \iff \|Mz\|_2 \neq 0 \iff$$
$$\|Mz\|_2^2 \neq 0 \iff z^TM^TMz \neq 0 \quad \forall z \neq 0$$

From which follows that M^TM is positive definite. □

Thus from proposition 1.0.1 it immediately follows that B^TB is positive definite. Let us now take a closer look at (1.2).

$$f(W_{out}) = \frac{1}{2}\|BW_{out} - Z\|_F^2 = \sum_{i=1}^{L+h}\sum_{j=1}^{m}\frac{1}{2}[\sum_{k=1}^{h}(b_{ik}w_{kj}) - z_{ij}]^2 \tag{1.3}$$

By taking the derivative w.r.t a specific $w_{k'j'}$ we obtain:

$$\frac{\partial \frac{1}{2}\|BW_{out} - Z\|_F^2}{\partial w_{k'j'}} = \sum_{i=1}^{L+h}(\sum_{k=1}^{h} b_{ik}w_{kj'} - z_{ij'})b_{ik'}$$

From which follows that the derivative w.r.t. a column $w_{:j'}$ of W_{out} is:

$$\frac{\partial \frac{1}{2}\|BW_{out} - Z\|_F^2}{\partial w_{:j'}} = B^TBw_{:j'} - B^Tz_{:j'} \tag{1.4}$$

Thus the derivative w.r.t. the full matrix is:

$$\nabla_{W_{out}} f(W_{out}) = \frac{\partial \frac{1}{2}\|BW_{out} - Z\|_F^2}{\partial W_{out}} = B^TBW_{out} - B^TZ \tag{1.5}$$

Before continuing, we first introduce some definitions useful for later

Definition 1.0.3 (**Convexity of a function**). *Let X be a convex subset of a real vector space, and let $f : X \to \mathbb{R}$ be a function. f is called convex if and only if for all $0 \leq t \leq 1$ and for all $x_1, x_2 \in X$:*

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

We can now prove the following.

Theorem 1.0.1 (**Convexity of ELM least square**). *The function $f(W_{out})$, as defined in (1.3), is convex.*

Proof. Let $\langle\cdot\rangle_F$ be the Frobenius product. By using the definition of Frobenius norm:

$$\begin{aligned} f(W) &= \frac{1}{2}\|BW - Z\|_F^2 \\ &= \frac{1}{2}\langle BW - Z, BW - Z\rangle_F \\ &= \frac{1}{2}\|Z\|_F^2 + \frac{1}{2}\|BW\|_F^2 - \langle Z, BW\rangle_F \end{aligned}$$

Let $W_1, W_2 \in \mathbb{R}^{h\times m}$, then:

$$\begin{aligned} \frac{1}{2}\|B(tW_1 + (1-t)W_2) - Z\|_F^2 &= \frac{1}{2}\|Z\|_F^2 + \frac{1}{2}\|B(tW_1 + (1-t)W_2)\|_F^2 \\ &\quad - \langle Z, B(tW_1 + (1-t)W_2)\rangle_F \\ &\leq \frac{1}{2}\|Z\|_F^2 + \frac{t}{2}\|BW_1\|_F^2 + \frac{1-t}{2}\|BW_2\|_F^2 \\ &\quad - \langle Z, B(tW_1 + (1-t)W_2)\rangle_F \end{aligned}$$

Which exactly satisfies the definition 1.0.3. □

Thus we conclude that the function we are trying to minimize is convex, and the minimum is found when the gradient in (1.5) is 0.

Actually, we can establish an even stronger result. To illustrate this, let's focus on a single column $w_{:j'}$ of W_{out}. By restricting our attention to this column, the problem simplifies to that of a quadratic multivariate function. Analyzing equation (1.4) it becomes evident that:

$$\nabla^2_{w_{:j'}} f(W) = B^T B \tag{1.6}$$

Thus the problem not only is convex, but also τ-convex and L-smooth. Here $\tau \leq \lambda_{min}$ and $L \geq \lambda_{max}$, where λ_{min} and λ_{max} are the smallest and highest eigenvalue of $B^T B$, respectively.

In this report we consider **two different approaches** in order to solve the least-square problem:

- A closed-form solution with the normal equations and Cholesky factorization;
- Nesterov accelerated gradient **(NAG)** as an algorithm of the class of accelerated gradient method.

In Chapter 2 we first focus on the two approaches used to solve the ELM least-square. We show the two algorithm, explain the behaviour of their parameters, and discuss any theoretical guarantees in terms of complexity (time and space) convergence (for the iterative algorithm) and stability (for the closed formula solution). Chapter 3 starts by describing the experimental setup of the problem, proceeding to discuss the empirical results obtained in the various experiments carried out. The last chapter is Chapter 4, where we summarize the results of our report.

Chapter 2

Algorithms

2.1 Nesterov accelerated gradient (NAG)

Nesterov Accelerated Gradient (NAG) enhances standard gradient descent by introducing a momentum component. Traditional gradient descent updates parameters based on the negative gradient of the loss function concerning the parameters, which can result in slow convergence. NAG improves this by incorporating a predictive element into the update process. Instead of immediately adjusting parameters using the current gradient, NAG first makes a tentative move in the direction of the previously accumulated gradient. It then calculates the gradient at this anticipated future position. This forward-looking approach enables NAG to correct its trajectory more accurately, resulting in more precise and quicker convergence.

Algorithm 1 *Nesterov accelerated gradient* (**NAG**).

Require: μ: Stepsize
Require: $\beta \in [0, 1)$: decay rate for momentum
Require: W_{in}: Randomly initialized fixed matrix
Require: $f(W)$: Stochastic objective function with parameters W
Require: W_0: Initial parameter matrix
Require: $\epsilon > 0$: Residual error on gradient norm
  $v_0 \leftarrow 0$ (initialize momentum)
  while $\|\nabla_{W_t} f(W_t)\|_F > \epsilon$ **do**
    $g_t \leftarrow \nabla_{W_t} f(W_{t-1} + \beta v_{t-1})$ (compute gradient after applying momentum)
    $v_t \leftarrow \beta v_{t-1} - \mu g_t$ (compute momentum + gradient descent)
    $W_t \leftarrow W_{t-1} + v_t$ (Update parameters)
  end while
  return W_t

Algorithm 1 describes the Nesterov accelerated gradient (NAG) method. The algorithm begins by initializing the momentum v_t to 0. In each iteration, it computes the gradient g_t at the anticipated future position, updates the momentum by combining the previous momentum scaled by β with the gradient descent step, regulated by a fixed stepsize μ, and then updates the model parameters accordingly. The parameter β controls the exponential decay rate of the momentum, progressively reducing the influence of older steps. The algorithm terminates when the gradient with respect to the current model parameters converges, indicated by the gradient norm falling below a specified threshold ϵ.

2.1.1 Considerations on NAG's parameters

In our initial presentation of the pseudocode for NAG we address two parameters: μ (the step size) and β (the momentum decay).

As stated in Chapter 1, the function we are trying to minimize is a multivariate quadratic function of the form:

$$q(w) = w^T B^T B w - \langle Bw, y \rangle + const$$

where w and y are vectors. It is important to note that all operations performed in NAG are element-wise. Consequently, the distinction between having vectors or matrices as model parameters is inconsequential. Therefore, we can consider W_{out} and Y as vectors without affecting the results. It is possible to demonstrate that, for specific values of μ and β, NAG converges linearly to the optimum. Assuming $\kappa = \frac{\tau}{L}$, where our function is τ-convex and L-smooth, we define the NAG process as follows:

$$\begin{cases} w_{k+1} = x_k - \frac{1}{L}\nabla q(x_k) \\ x_{k+1} = w_{k+1} + \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}(w_{k+1} - w_k) \end{cases}, \quad \text{with } \beta = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}, \quad \mu = \frac{1}{L} \tag{2.1}$$

For this specific choice of μ and β, the following can be demonstrated:

Theorem 2.1.1. *The sequence $\{q(x_k)\}_{k\in\mathbb{N}}$ produced by NAG on a τ-convex L-smooth function satisfies*

$$q(w_k) - q^* \leq \frac{\tau + L}{2}\|x_0 - x^*\|_2^2 exp(\frac{-k}{\sqrt{\kappa}})$$

with $q(w^*) = q^*$

Meaning that the algorithm, if we stop when ϵ-accuracy is achieved:

$$\frac{L+\tau}{2}\left(1 - \frac{1}{\sqrt{\kappa}}\right)^k \|x_0 - x^*\|_2^2 \leq \epsilon$$
$$k \geq \sqrt{\kappa}\log\left(\frac{1}{\epsilon}\right) + const$$

thus achieving a linear convergence rate of $O(\sqrt{\kappa}\log\left(\frac{1}{\epsilon}\right))$. However, a significant challenge arises: obtaining the values of τ and L efficiently is not feasible, as we would need to compute the eigenvalue decomposition of the matrix $B^T B$, which requires $O(h^3)$ time. This time complexity is so high that it would allow us to solve the problem directly using a closed-form solution, making the iterative process we are employing redundant.

A straightforward method to address this issue involves treating μ and β as fixed hyper-parameters. By performing a grid search, we can determine the optimal values for these parameters, along with ϵ (the tolerance), which needs to be initially tested with different values independently μ and β.

We also propose another alternative to this approach, computing the stepsize for μ as follows:

$$\mu_k = \arg\min_{\mu>0} x_k - \mu\nabla q(x_k) = \frac{\|\nabla q(x_k)\|_F^2}{tr\left(\nabla q(x_k)^T B^T B \nabla q(x_k)\right)}$$

Here, $tr(\cdot)$ denotes the trace function, which is the sum of the diagonal elements of the matrix, treating the gradient as a vector rather than as a matrix. This step size is derived from the exact line search, which guarantees linear convergence for standard gradient descent without momentum. Therefore, in the worst-case scenario, we achieve linear convergence $O(\kappa log\left(\frac{1}{\epsilon}\right))$ for $\beta = 0$, although the convergence should be faster when using $\beta \neq 0$.

Another approach involves leveraging the fact that, as discussed in Chapter 1, the problem we are trying to minimize is the sum of m quadratic functions, each of which is independent and parameterized by a column $w_{:j}$ of W_{out}. Consequently, we could perform an exact line search for each of these m free parameter vectors, updating each $w_{:j}$ with its respective optimal stepsize.

Utilizing exact line search allows for the benefit of employing a larger step size when appropriate, avoiding the constraint that a fixed step size must remain sufficiently small across all iterations. However, with a fixed step size, each iteration involves two gradient evaluations: one for checking convergence and another for performing the gradient descent step. Each gradient evaluation is computationally intensive, costing $O(h^2m)$, in contrast to the lower cost of other element-wise operations, which is $O(hm)$.

Instead, line search requires computing the step size through two matrix products, each with a time cost of $O(h^2m)$ (quadratic in h, where $h >> m$). This includes one computation for the numerator and one for the denominator. Given that line search requires just two additional $O(h^2)$ operations per iteration, we expect that the computational cost per iteration should not increase significantly. Moreover, the total cost of the iterative process should substantially decrease due to the reduced number of iterations required for convergence.

Thus far, we have discussed methods for optimizing the step size to eliminate it as a hyperparameter. However, can we apply a similar strategy to β? While the optimal value of β can be fixed when working with strongly convex functions, it is common practice to use a fixed scheduling of β for convex but not strongly convex functions. The need for scheduling arises because the optimal β for convex functions follows a specific schedule rather than a fixed value. A practical and well-established scheduling approach[2] is as follows:

$$\theta_0 = 1$$
$$\theta_{k+1} = \left(\frac{\sqrt{1 + 4(\theta_k)^2}}{2}\right)$$
$$\beta_{k+1} = \frac{\theta_k - 1}{\theta_{k+1}}$$

2.1.2 Alternatives to accelerated gradient descent

Method	Quadratic function	Convex smooth function
Standard gradient descent	$O(\kappa log(\frac{1}{\epsilon}))$	$O(\frac{1}{\epsilon})$
Conjugate gradient	$O(\sqrt{\kappa} log(\frac{1}{\epsilon}))$	n-step quadratic
NAG	$O(\sqrt{\kappa} log(\frac{1}{\epsilon}))$	$O(\frac{1}{\sqrt{\epsilon}})$
Adam[3]	/	$O(\frac{1}{\epsilon^2})\star$

Table 2.1: Main gradient descent algorithms for solving convex optimization. $\kappa = \frac{L}{\tau}$. $\star$ Adam convergence is for a smooth convex stochastic function

When deciding which iterative algorithm to use, we initially considered using the Adam optimizer to solve our optimization problem. However, after evaluating the properties of various optimization algorithms, as detailed in Table 2.1, we realized that Adam might not be the most suitable choice for our relatively straightforward problem, since our task involves finding the global minimum of a smooth and strongly convex function. In our case study we could not establish a specific bound for Adam's performance, however other methods such as SGD and Adagrad are known to converge linearly[4, 1], although at a slower rate compared to NAG or the Conjugate Gradient.

The benefits of NAG for our problem are significant. NAG demonstrates superior convergence compared to standard gradient descent and performs comparably to the Conjugate Gradient method, making it a highly effective approach. Assuming $\kappa = \frac{\lambda max}{\lambda min}$, for smooth strongly convex functions, NAG achieves a convergence rate of $O(\sqrt{\kappa} log(\frac{1}{\epsilon}))$ which is better than the standard gradient descent boundary of $O(\kappa log(\frac{1}{\epsilon}))$. Additionally, NAG's convergence rate is arguably better than the Conjugate Gradient method, which converges at n-step quadratic rate:

$$\left\|x^{i+n} - x^*\right\| \leq r\left\|x^i - x^*\right\|^2$$

Although this is generally accurate, the primary reason for selecting NAG w.r.t. conjugate gradient in our project was the requirement to choose an algorithm from the accelerated gradient descent family.

2.2 Closed formula solution

Before introducing the Cholesky decomposition, we first need to explain how we can reach a closed formula solution for our problem. By imposing (1.5) equal to 0 we obtain:

$$\begin{aligned} &B^T B W_{out} - B^T Z = 0 \quad \iff \\ &B^T B W_{out} = B^T Z \quad \iff \\ &(A^T A + \alpha I) W_{out} = A^T Y, \end{aligned} \tag{2.2}$$

where (2.2) is the normal equation relative to our problem. Since the coefficient matrix remains the same for subsequent linear systems, it is sufficient to define the decomposition once fo $A^T A - \alpha I$.

Applying the Cholesky decomposition to $M = (A^T A + \alpha I)$ we obtain:

$$(A^T A + \alpha I) = LL^T,$$

where the elements of L are computed as follows:

1. $l_{11} = \sqrt{m_{11}}$, which is well-defined by Prop 1.0.1 as $m_{11} > 0$ as M is positive definite;
2. $L_{ii} = \sqrt{m_{11} + \sum_{k=1}^{i-1} L_{ik}^2}$;
3. $L_{ij} = \frac{1}{L_{jj}}(\sqrt{m_{ij} \sum_{k=1}^{j-1} L_{ik} L_{jk}})$

At the end of the algorithm we obtain L^t and L, with $(A^T A + \alpha I) = LL^T$.

At this point we can solve the m independent linear systems (one per column of W_{out})

$$LL^T W_{out} = A^T Y.$$

Since L is a lower triangular matrix with a nonzero diagonal (ensuring it is invertible), we can efficiently solve for W_{out} using forward and backward substitution.

2.2.1 The Cholesky–Banachiewicz algorithm

The Cholesky decomposition algorithm decomposes a symmetric positive definite matrix M into the product of a lower triangular matrix L and its transpose L^T, such that $A = LL^T$. In the explanation of the algorithm $A^T A + \alpha I = B^T B = M$ that is a matrix of dimensionality $h \times h$.

Since M is symmetric and positive definite, we can express it in the form $M = LL^T$, where $L \in \mathbb{R}^{h \times h}$ is a lower triangular matrix. Then, assuming:

$$L = \begin{pmatrix} L_{11} & 0 & 0 & \cdots & 0 \\ L_{21} & L_{22} & 0 & \cdots & 0 \\ L_{31} & L_{32} & L_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{h1} & L_{h2} & L_{h3} & \cdots & L_{hh} \end{pmatrix}$$

We obtain that:

$$M = LL^T = \begin{pmatrix} \sum_{k=1}^{1} L_{1k}^2 & \text{symmetric} & \cdots & \text{symmetric} & \\ \sum_{k=1}^{1} L_{2k}L_{1k} & \sum_{k=1}^{2} L_{2k}^2 & \cdots & \text{symmetric} & \\ \sum_{k=1}^{1} L_{3k}L_{1k} & \sum_{k=1}^{2} L_{3k}L_{2k} & \sum_{k=1}^{3} L_{3k}^2 & \cdots & \\ \vdots & \vdots & \vdots & \ddots & \\ \sum_{k=1}^{1} L_{hk}L_{1k} & \sum_{k=1}^{2} L_{hk}L_{2k} & \sum_{k=1}^{3} L_{hk}L_{3k} & \cdots & \sum_{k=1}^{h} L_{hk}^2 \end{pmatrix}$$

From this equation, we can then express L in terms of M:

$$L = \begin{pmatrix} \sqrt{M_{1,1}} & 0 & 0 & \cdots & 0 \\ \frac{M_{2,1}}{L_{1,1}} & \sqrt{M_{2,2} - L_{2,1}^2} & 0 & \cdots & 0 \\ \frac{M_{3,1}}{L_{1,1}} & \frac{M_{3,2} - L_{3,1}L_{2,1}}{L_{2,2}} & \sqrt{M_{3,3} - L_{3,1}^2 - L_{3,2}^2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{M_{h,1}}{L_{1,1}} & \frac{1}{L_{2,2}}\left(M_{h,2} - L_{h,1}L_{2,1}\right) & \frac{1}{L_{3,3}}\left(M_{h,3} - L_{3,1}L_{h,1} - L_{3,2}L_{h,2}\right) & \cdots & \sqrt{M_{h,h} - \sum_{k=1}^{h-1} L_{h,k}^2} \end{pmatrix}$$

obtaining that:

$$L_{ii} = \sqrt{M_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \qquad \text{for } i = 1, 2, \ldots, h \quad \text{(diagonal elements)}$$

$$L_{ij} = \frac{1}{L_{jj}} \left(M_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk} \right) \qquad \text{for } i > j \quad \text{(off-diagonal elements)}$$

Algorithm Steps

Based on the recursive formulas just obtained, the entries of the lower triangular matrix L can be computed in a row-wise manner. Specifically, starting from the first row, each element $L_{i,j}$ is determined using only the previously computed entries of L and the corresponding entries of the input matrix M. Since the computation of $L_{i,i}$ (diagonal elements) depends solely on earlier elements in the same row, and the computation of $L_{i,j}$ for $i > j$ (off-diagonal elements) relies on already computed values in previous columns, the construction of L proceeds from top to bottom and left to right. What follows are the steps and pseudocode of the algorithm used.

1. **Input:** The matrix A of dimension $L \times h$, a target matrix Y of dimension $L \times m$ and a coefficient α.
2. **Output:** A lower triangular matrix L such that $M = LL^T$.
3. **Initialization:** Create an $h \times h$ matrix L filled with zeros.
4. **Computation:** Compute $(A^TA + \alpha I)$ and A^TY then for each element (i, j) of L with $1 \leq j \leq i \leq h$:

(a) If $i = j$ (diagonal elements):

$$L_{ii} = \sqrt{M_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

(b) If $i > j$ (off-diagonal elements):

$$L_{ij} = \frac{1}{L_{jj}} \left(M_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right)$$

Pseudocode

The algorithm can be expressed in pseudocode as follows:

```
Input: Matrix A of dimension L x h, coefficient alpha
Output: Lower triangular matrix L
1.Compute A^t
2.M= A^T A + alpha I
3.Initialize L as a zero matrix of dimension h x h
4. For i from 1 to h:
   5. For j from 1 to i:
      6. If i == j:
         7. L[i][i] = sqrt(M[i][i] - sum(L[i][k]^2 for k from 1 to i-1))
      8. Else (i > j):
         9. L[i][j] = (M[i][j] - sum(L[i][k] * L[j][k] for k from 1 to j-1)) / L[j][j]

10. Return L
```

2.2.2 Solving the linear systems

Once we define the algorithm to obtain the Cholesky Decomposition we can now execute it to obtain the matrix L. We then need to solve the following system:

$$LL^T W_{out} = A^T Y$$

Since each column of W_{out} corresponds to a column of the target matrix $A^T Y$, we solve the system for each column separately. Let w_i represent a column of W_{out} and y_i represent the corresponding column of $A^T Y$. For simplicity, we generalize this notation as w and y, leading to the system:

$$LL^T w = y$$

Given the triangular structure of L, we solve this system efficiently using the following steps:

1. Solve $Lz = y$ for z. (Forward Substitution)
2. Solve $L^T w = z$ for w. (Backward Substitution)

The final solution W_{out}^* is obtained by concatenating all the solutions w found by solving the m linear systems.

2.2.3 Algorithm Complexity

Matrix multiplication

Before we apply the Cholesky decomposition we have to define the initial matrices M and A^TY.

1. **Compute** A^TA: A^T is the transpose of A, which has dimensions $L \times h$. Multiplying A^T (of dimensions $h \times L$) by A results in a matrix of dimensions $h \times h$.

The complexity of this multiplication is:

$$O(h \cdot L \cdot h) = O(L \cdot h^2)$$

2. **Compute** αI:

I is the identity matrix of dimensions $h \times h$. Multiplying the scalar α by the identity matrix has a complexity of:

$$O(h)$$

3. **Add the matrices** A^TA and αI:

The addition of two $h \times h$ matrices has a complexity of:

$$O(h^2)$$

4. **Compute** A^TY:

A^T has dimensions $h \times L$, and Y has dimensions $L \times m$. The matrix multiplication results in a complexity of:

$$O(h \cdot L \cdot m)$$

Thus, assuming $h >> m$ the overall complexity of this first phase is dominated by the complexity of the matrix multiplication A^TA.

Time complexity of the Cholesky decomposition

To obtain the Cholesky decomposition, we have to compute:

1. *For diagonal elements* L_{ii}:

$$L_{ii} = \sqrt{M_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

This involves $i - 1$ operations for the summation and one square root.

2. *For off-diagonal elements* L_{ij} (where $i > j$):

$$L_{ij} = \frac{1}{L_{jj}} \left(M_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right)$$

This involves $j - 1$ operations for the summation and one division.

The total number of operations is the sum of the operations for all elements:

$$\sum_{i=1}^{h} \sum_{j=1}^{i} (2j - 1)$$

This double summation can be simplified to:

$$\sum_{i=1}^{h} \left(\frac{i(i+1)}{2} \right) = \frac{1}{2} \sum_{i=1}^{h} i^2 + \frac{1}{2} \sum_{i=1}^{h} i$$

Using the known summations:

$$\sum_{i=1}^{n} i^2 = \frac{h(h+1)(2h+1)}{6}$$

$$\sum_{i=1}^{h} i = \frac{h(h+1)}{2}$$

We combine these results:

$$\frac{1}{2}\left(\frac{h(h+1)(2h+1)}{6} + \frac{h(h+1)}{2}\right) = \frac{1}{2}\left(\frac{h(h+1)(2h+1+3)}{6}\right) = \frac{1}{2}\left(\frac{h(h+1)(2h+4)}{6}\right) = O(h^3)$$

The computational complexity of the Cholesky decomposition algorithm is $O(h^3)$. This makes the algorithm efficient for moderate-sized matrices but potentially computationally expensive for very large matrices. The overall complexity depends on the chosen h, the dimension of the hidden layer, which is greater than L (the number of samples) in most cases, although we analyze both scenarios.

Time complexity for the resolution of linear systems

To solve the system $LL^T w = y$ using forward and backward substitution, assuming L is a lower triangular matrix of size $h \times h$:

1. Forward Substitution for $LZ = Y$

   In each iteration i (where i ranges from 1 to h), we compute the value of w_{ij} using the previously computed values. The total cost is given by:

   $$\sum_{i=1}^{h} i = \frac{h(h+1)}{2} = O(h^2)$$

   As we have to apply again this method for each column of Z, since it has m column we obtain a total cost of:

   $$m \cdot O(h^2) = O(mh^2)$$

2. Backward Substitution for $L^T W_{out} = Z$

   Similarly, in each iteration i (where i ranges from 1 to h), we compute the value of w_{ij} using the subsequent values. The total cost to obtain W_{out} is:

   $$m \cdot O(h^2) = O(mh^2)$$

Combining both steps, the overall computational complexity for solving the system $LL^T W_{out} = A^T Y$ using forward and backward substitution is $O(h^2)$, which is clearly less computationally intensive than the matrix multiplication and the Cholesky decomposition

Algorithm Stability

Since M is a positive definite matrix, its Cholesky decomposition is stable without pivoting. Solving the two linear systems is also a backward stable operation. However, this does not guarantee the backward stability of the entire process, as the matrix product $A^T A$ is not inherently backward stable. In fact, while we can manipulate the matrix W_{in} to be any matrix we want, the data matrix X is fixed and we cannot make any assumption on its structure.

2.3 Final considerations on the algorithms

For both algorithms, the matrix product B^TB can be computed in $O(L^2h)$ time. Each iteration of NAG has a computational cost of $O(h^2)$, while the closed-form solution has a cost of $O(h^3)$. When $L >> h$ the matrix product becomes the most expensive operation, implying that the overall computational time for both algorithms won't differ too much; but in the more realistic case of $h >> L$ the actual runtime of the algorithms becomes more significant, as it constitutes the more costly operation.

Other factors that impact the performance of the two algorithms when compared to each other include the regularization parameter α and the stopping criteria tolerance ϵ for the accelerated gradient.

The tolerance ϵ significantly impacts the performance of the accelerated gradient algorithm. Ideally, since our objective to reach a solution that is as close to the global optimum as possible, we would like to choose an ϵ that stops the algorithm when the distance from the true optimum of the solution found is in the order of machine precision.

The regularization parameter α also impacts the performance of the algorithms. When using the closed-form solution, a higher α lowers the condition number of B^TB, which can improve numerical stability. However, this does not affect the product A^TA, which remains the primary bottleneck in terms of solution quality. In contrast, for the accelerated gradient method, a higher regularization leads to faster convergence to the optimum. This is because the number of iterations in gradient descent is influenced by the condition number of the matrix, and a lower condition number results in fewer iterations needed for convergence.

Chapter 3

Experiments

3.1 Experimental setup

The Dataset used for the experiments is the Machine learning course ML CUP dataset of years 2023/2024, containing a total of 1000 samples. Each sample is a vector of 10 features $x \in \mathbb{R}^10$, and each target is a vector of 3 features $y \in \mathbb{R}^3$. Fig. 3.1 shows an example of the dataset structure.

We initialize the ELM by sampling each matrix from a normal distribution with zero mean and a standard deviation equal to the square root of the fan-in of the matrix. Specifically, for each weight matrix W, the elements are initialized as:

$$W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{1}{\text{fan-in}}}\right)$$

where fan-in denotes the number of input units (rows) to the layer associated with W.

All the experiments were executed on a Apple M1 8-core cpu with 8GB of RAM.

3.2 Considerations on matrix conditioning

Before discussing the results of the experiments for NAG and the Cholesky decomposition approach, we want to focus for a moment on eventual problems that can be caused by numerical error. In fact, as we had previously mentioned in Chapter 2 both approaches are impacted particularly by the conditioning of the $B^T B$ matrix (For NAG in the convergence speed, and for Cholensky in the numerical precision due to the matrix product).

Fig. 3.2 shows the conditioning of some of the matrices involved in the model operations. Since W_{in} and A are rectangular matrices, we define the condition number in this context as the ratio between the highest and smallest singular value. Assuming a generic rectangular matrix $C \in \mathbb{R}^{\gtrless \times \lessgtr}$ we have:

$$\text{cond}(C) = \frac{\sigma_1(C)}{\sigma_{\min(m,n)}(C)}$$

	ID	x1	x2	x3	x4	...	x9	x10	y1	y2	y3
0	1	-0.917280	-0.712727	-0.989904	0.992819	...	-0.688548	0.616890	7.897453	-35.936382	21.077147
1	2	-0.858784	0.998755	-0.998396	0.999909	...	0.661759	-0.800155	-9.330632	19.901571	6.069154
2	3	-0.990441	0.958726	-0.998675	0.997216	...	-0.684630	0.922901	14.849400	3.374090	19.667479
3	4	0.937117	0.984474	-0.612420	0.999812	...	-0.921444	-0.974766	-46.591854	13.734777	17.953600
4	5	-0.906628	-0.884567	-0.932487	0.941037	...	-0.508268	0.691798	8.217500	-45.885254	14.894251

Figure 3.1: An example of the dataset structure.

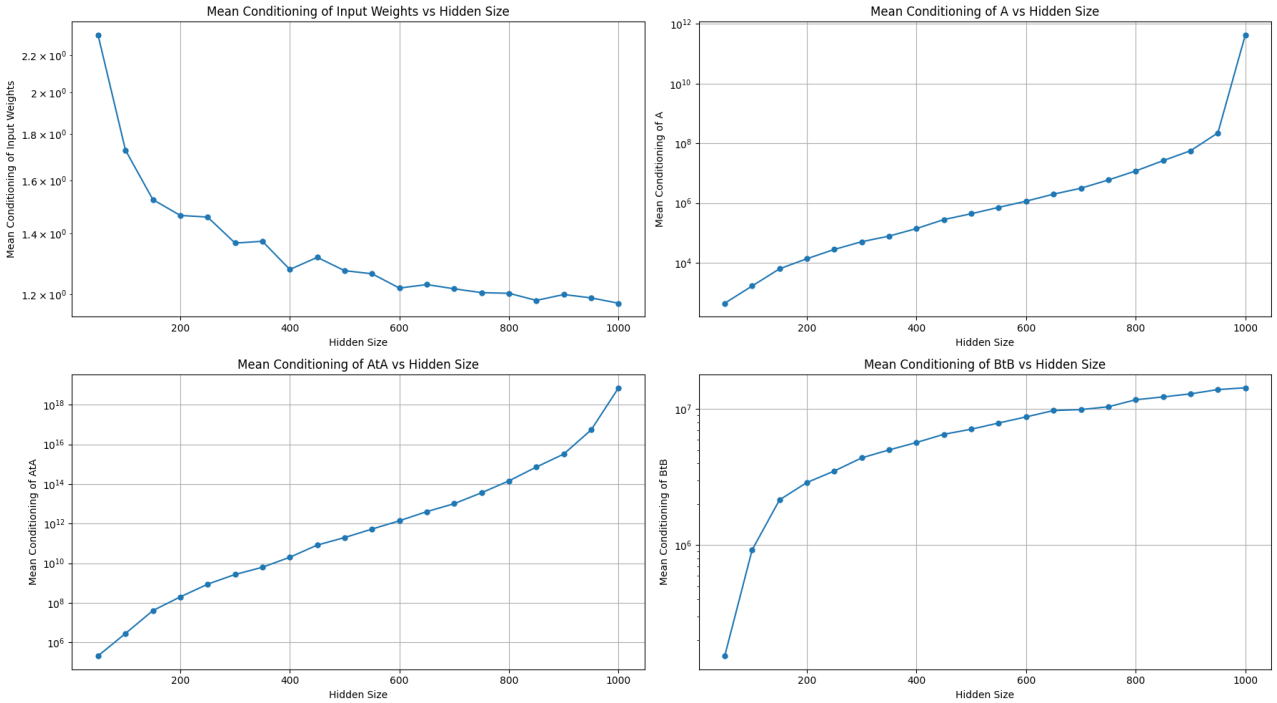


Figure 3.2: Average conditioning for different matrices and model sizes. The results are the average over 5 experiments.

where σ_1 and $\sigma_{\min(m,n)}$ are the highest and smallest singular values

We can first observe that the conditioning of W_{in} is low, between 2.5 and 1 in general, regardless of the model size. Considering that the conditioning of the data matrix X is ~ 7.8, one would expect the conditioning of $A = \sigma(XW_{in}))$ to be in a similar order of magnitude. However, this is not clearly the case, as it is shown by the top-right plot, where the conditioning is already relatively high for only 50 hidden units (first datapoint) and progressively increases up to 10^{12} for 1000 hidden units.

Of course, this problem is getting even more accentuated, as for both algorithms we need to compute A^TA, which in the worst case scenario of 1000 hidden units we observe a condition number in the order of 10^{18}. However, as we need to sum αI to A^TA in our Least squares regularized problem, you can see that in practice we tend to have conditioning between 10^5 and 10^7 for B^TB (bottom-right plot).

We determined that the main fault of this behavior stems from the product XW_{in}. The problem resides in the fact that, while it can be proven that for $m \geq n \geq p$ the product between two rectangular matrices $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$ is bounded by:

$$\text{cond}(AB) = \frac{\sigma_1(AB)}{\sigma_{\min(m,p)}(AB)} \leq \frac{\sigma_1(A)}{\sigma_{\min(m,n)}(A)} \cdot \frac{\sigma_1(B)}{\sigma_{\min(n,p)}(B)} = \text{cond}(A) \cdot \text{cond}(B).$$

and we leave the proof in Sec. A.1, this is not the case when $m \geq n \leq p$, which is what happens when we use more than 10 units in the hidden layer, as X has 10 columns. What happens is that we cannot put a lower bound on the decrease of the smallest eigenvalue, as it is shown in Fig. 3.3.

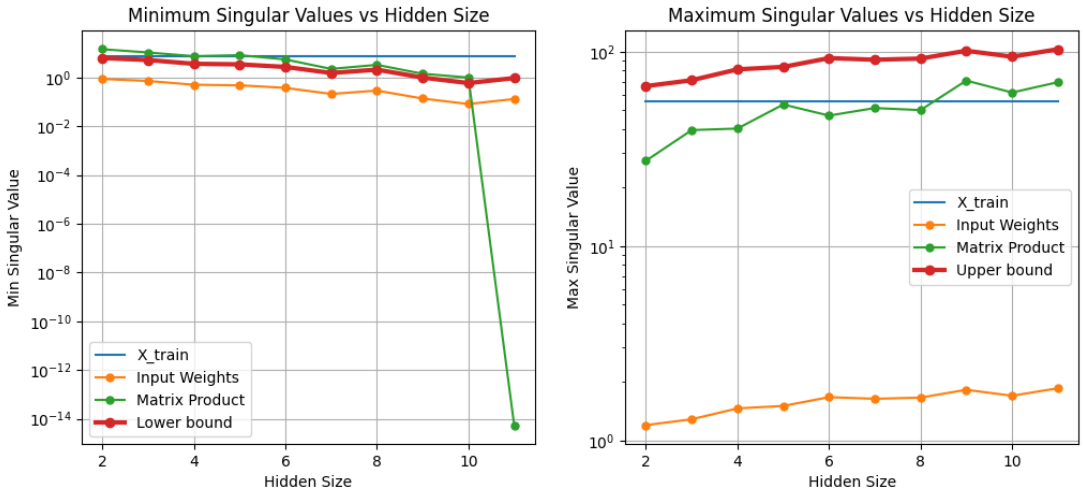


Figure 3.3: (left) smallest singular value for different model size. (right) highest singular value for different model sizes. As long as the number of units is less than the amount of features in the dataset (10) the singular values of the matrix product are bounded, but when we start having more hidden units (11) we immediately observe the lower bound on the smallest singular value is not respected anymore.

3.3 Experimental results

3.3.1 Experiments on NAG

In Chapter 2 we discussed various heuristics that could be employed to solve our problem using NAG. Our primary focus is on analyzing the convergence speed of these alternatives, comparing them in terms of both the number of iterations and the total runtime of the algorithm.

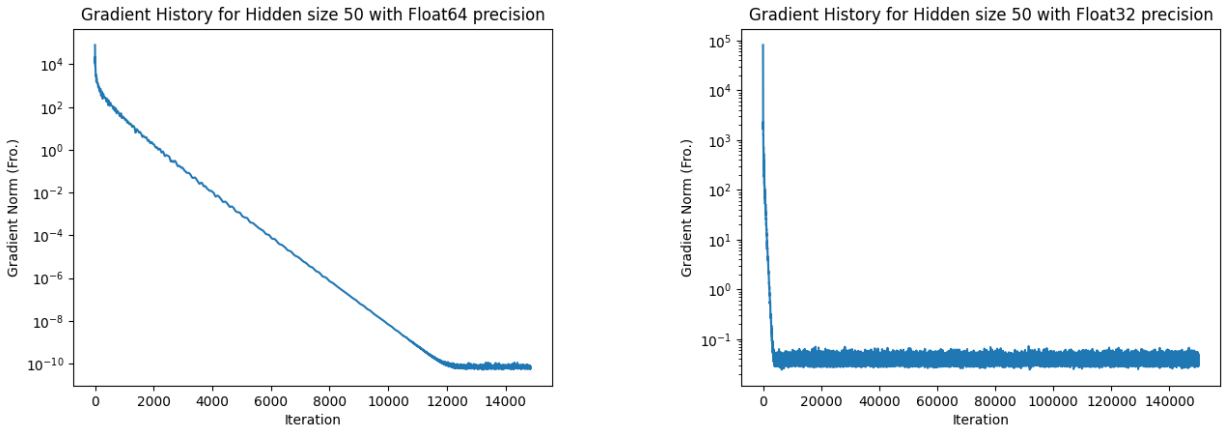
Trough our analysis we will inspect the impact of the different parameters that are present in NAG, namely:

- The stopping criterion, regulated by ϵ;
- The regularization coefficient, regulated by α;
- The learning rate/stepsize of the gradient descent μ;
- The momentum coefficient β;

We inspect in all experiments models with 50,100, 500 and 1000 hidden units. For every of the following sections about NAG all experiment were stopped at *one million iterations* in order to prevent too long runs either due to errors in how we set the parameters or to simply slow convergence speed of some configurations.

Impact of ϵ

To investigate the convergence of NAG, the first problem we incurred into when initiating our analysis was which ϵ value to choose. In order to assess the correct behavior of of the algorithm, we wanted ϵ as low as possible, as the lower it is, the closer we are to the optima of our problem. However it is important to denote the presence of machine precision: as we employed double format for machine precision, it is impossible to ask to stop when the gradient is below 10^{-16} due to machine precision.



(a) (b)

Figure 3.4: (a) Example of algorithm convergence with double machine precision (b) Example of algorithm convergence with float32 machine precision.

Let us look at Figure 3.4, showing the gradient (Frobenius norm) for each iteration of NAG with 50 hidden units and $\alpha = 10^{-2}$. What we can observe is that, if left unattended, the algorithms seem to converge linearly until reaching a certain value, different between the two floating point formats. What happened is that the updates to the model's output weight matrix have in fact reached the machine precision, and thus the algorithm is unable to further converge towards the optimum of the function.

This is mainly motivated by two observations. The first one is that, by observing our example, the optimal stepsize (as defined by Eq. 2.1) is in the order of 10^{-5}, which means that updates to the weight matrix are in the order of $10^{-5} \times 10^{-11} = 10^{-16}$ for double precision format, meaning that they are very close to machine precision. The second observation instead is that this behavior for float32 precision format happens around 10^{-2}, meaning there is a ratio of 10^{-9} between the stationarity point of float32 w.r.t double precision, which is exactly the same ratio there is between the machine precision of float32 ($\sim 10^{-7}$) and the one of double format ($\sim 10^{-16}$) further validating our hypothesis.

In practice, we found that for $\alpha = 10^{-2}$ with models with at least 500 units $\epsilon = 10^{-9}$ runs usually show little to no presence of this machine precision instability, while for less than 500 units we use $\epsilon = 10^{-10}$.

Convergence analysis for different stepsizes

After having established the values of ϵ used for future runs, we proceed to investigate the convergence of NAG when changing the stepsize used. For this section, we fix $\alpha = 10^{-2}$ and optimal β (as per Eq. 2.1) in order to make the execution of gradient descent reasonably fast regardless of the stepsize employed.

For clarity of notation, throughout the plots presented in this work, the label "auto" for the step size refers to the use of exact line search. In contrast, the label "col" indicates that the exact line search is performed independently on the columns of the W_{out} matrix.

Let us first look at Fig. 3.5 which shows the convergence behavior of the gradient for different model sizes. As it is possible to observe, the plots show what seems linear convergence, regardless of the specific stepsize. Of course, the further lower we go from the optimal stepsize of each model, the slower the algorithm will reach the optimum, as it is shown in all plots. In particular, it is possible to assess that indeed for the values of ϵ chosen the algorithm manages to correctly stop when the gradient is smaller than ϵ in Frobenius norm. Moreover, it is interesting

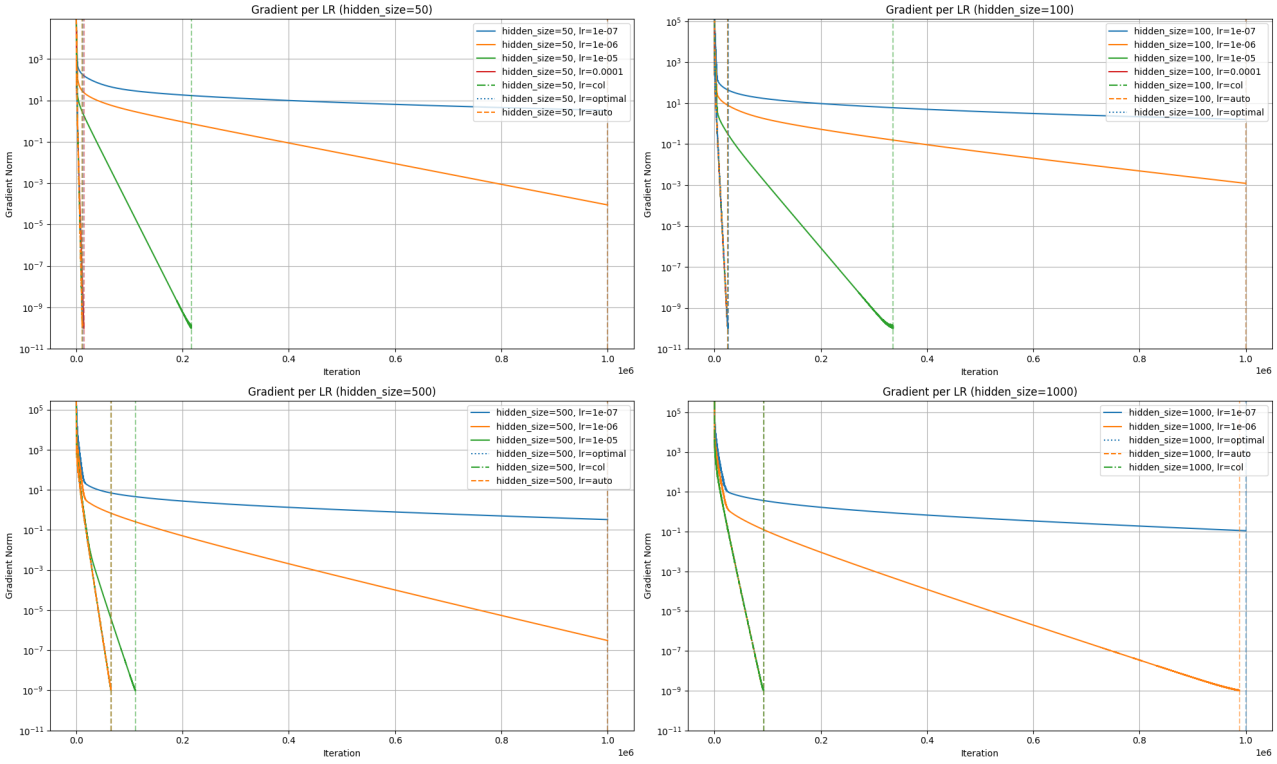


Figure 3.5: Gradient convergence for different model sizes and stepsizes.

to see how the exact line search manages to converge with a speed that this comparable to the one with optimal values.

What also could convince us of the correct convergence of NAG is shown in Fig. 3.6, where we measure the distance (in Frobenius norm) of the output weight matrix found by NAG from the output weight matrix found by using QR factorization. The reason we compare it against QR factorization, instead of directly comparing it to the solution found by using the Cholesky decomposition , is that the QR factorization tends to yield a solution that is closer to the minimum (as the algorithm is backwards stable) compared to Cholesky, mitigating the impact of the ill-conditioning of the matrices.

We can make two main observations about the plots in Fig. 3.6. The first observation is that indeed what we are seeing is a linear convergence to the optimal solution, regardless of the stepsize. In particular, the closer the stepsize is to the optimal one, the faster the convergence is in terms of iterations. The second observation is instead a less fortunate one: the larger the model is the more far away we are from the optimal solution. This is obviously a consequence of what we already stated in the previous section, as the bigger the model is, the lower is the optimal stepsize and the higher is ϵ.

However, while the previous plots looked promising, the reality is much harsher. Let us look at Fig. 3.7, where we show the relative difference between the Least Square Error (which is our function value) obtained by NAG with fixed stepsize (notice the absence of optimal stepsize and line search curves from the plot) and the Least Square Error of the QR solution. While for the most part we observe a linear decrease, for 100 hidden units with $\mu = 10^{-4}$ we do observe a strange trend.

Fig. 3.8 shows the relative difference of NAG with optimal parameters against the optimal solution. We observe that for relatively small models (50 and 100 hidden units) the behavior shown is a linear convergence, as expected. However, for big, worse conditioned models, this configuration shows an oscillatory behavior. We mainly deem the ill-conditioning of bigger models as the main cause of this problem, as for obtaining optimal μ and β we need to compute

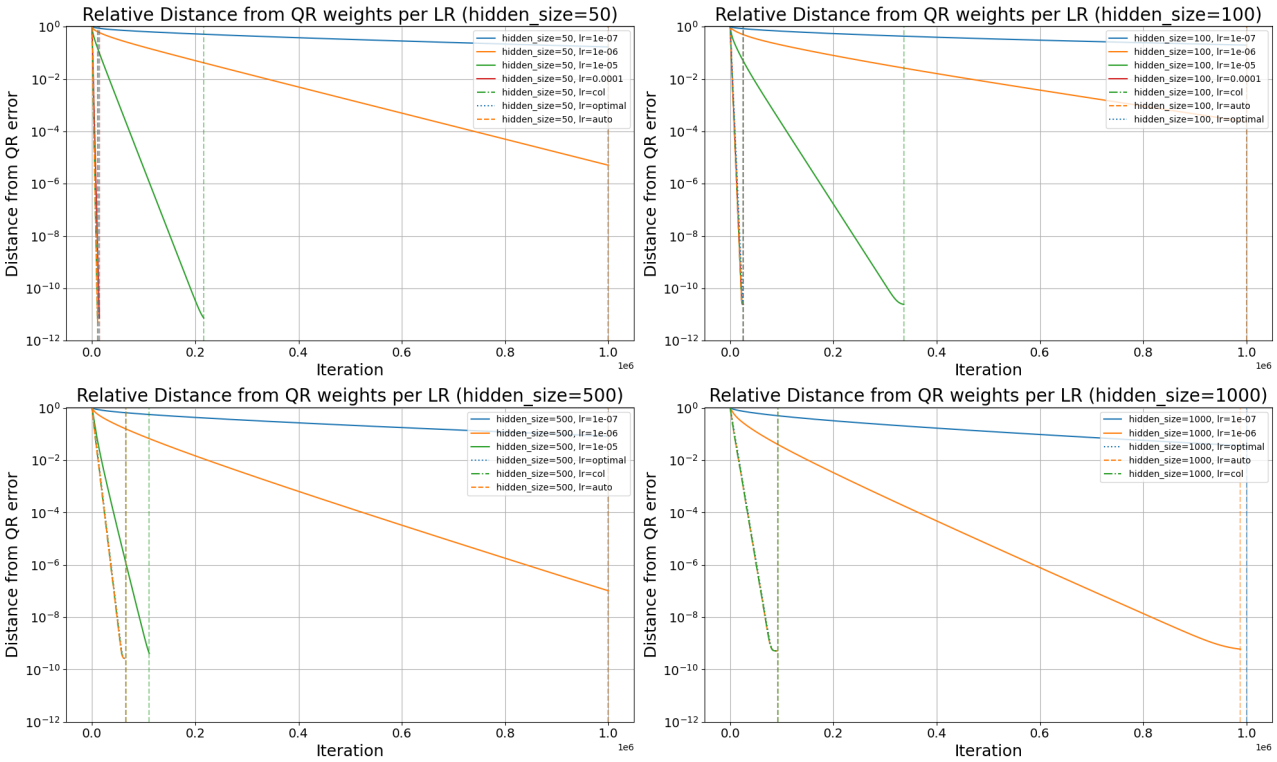


Figure 3.6: Relative distance from solution found by employing QR factorization. The distance is expressed using Frobenius norm.

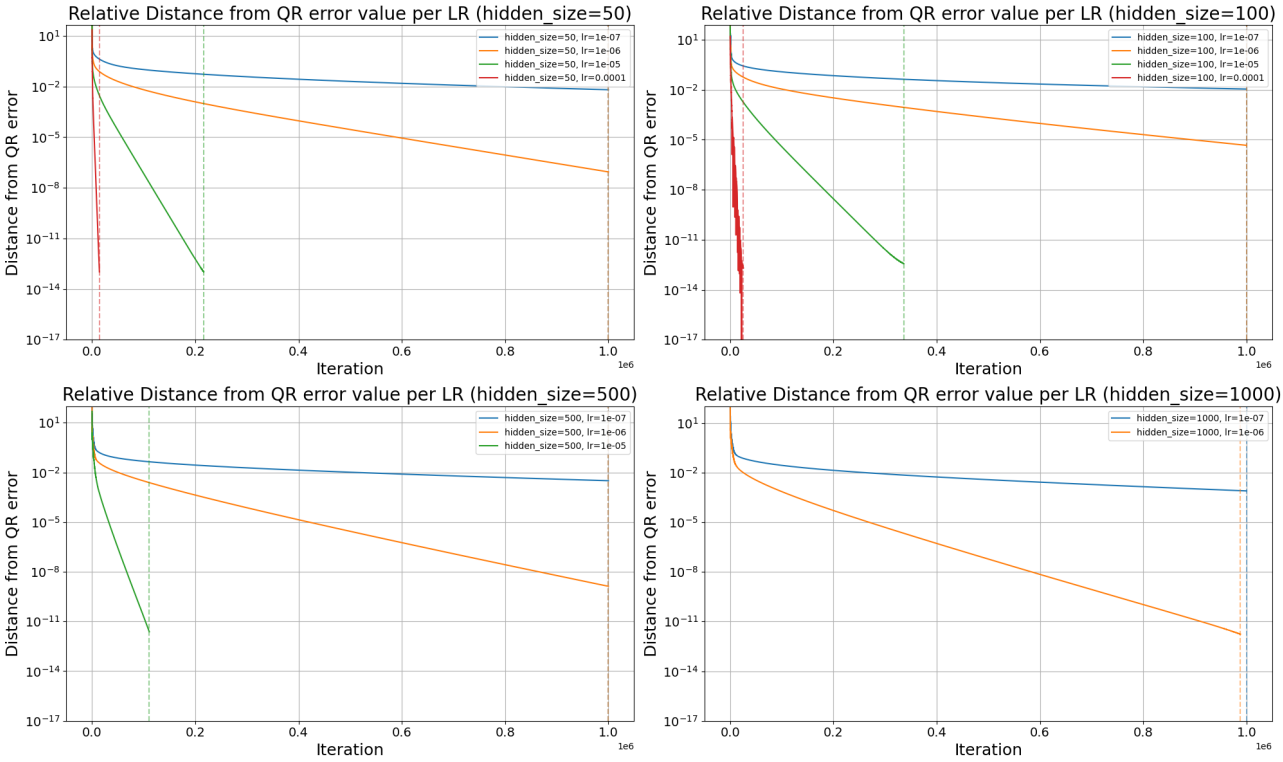


Figure 3.7: Relative difference between LSE of NAG and the LSE of the solution with QR factorization.

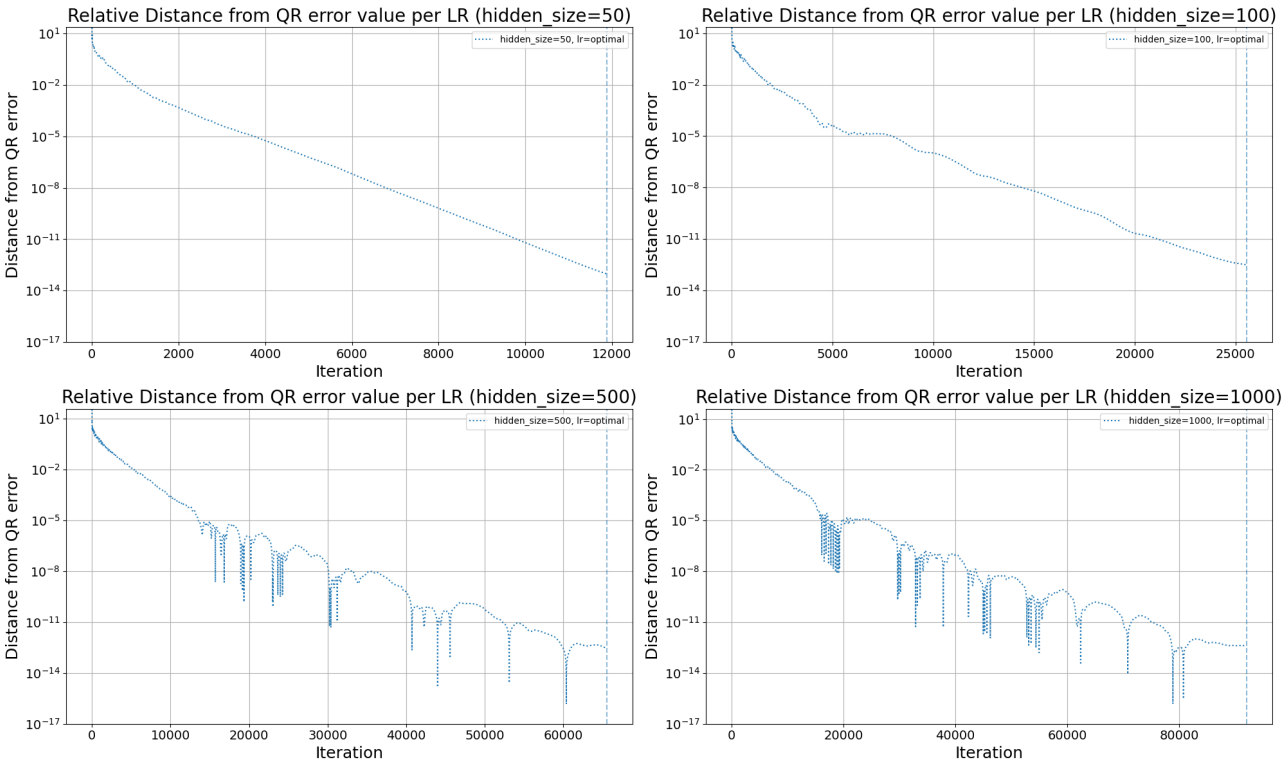


Figure 3.8: Relative difference of the LSE of NAG with optimal β and μ and LSE of QR factorization.

the eigenvalues of B^TB, and in doing so we assume that this matrix is symmetric, while in practice this is not the case due to numerical errors.

This behavior is caused by having too large values of β and μ, specifically higher than the (true) optimal values. Indeed, this is also the case for the 10^{-4} stepsize with 100 hidden units, as the optimal stepsize in that case is $\sim 7\times10^{-5} < 10^{-4}$. This is also justified by the previously shown Fig. 3.7, as it is possible to observe that for low enought fixed stepsizes we get a linear decrease of the LSE (for instance, for 1000 units we obtain a final LSE that is $\sim 10^{-11}$ off from the optimum).

For exact line search, presented in Fig. 3.9, the situation is quite similar. The observed behavior is counterintuitive: rather than exhibiting a straightforward linear convergence pattern, the curve demonstrates an oscillatory nature. Initially, the difference is positive, as the gradient descent is still far from the optimum. As the algorithm progresses, NAG approaches the optimum, reducing the absolute difference. However, the difference subsequently increases in magnitude again, this time becoming negative, suggesting that the solution provided by NAG now surpasses that of the QR factorization in terms of LSE. While the general trend, ignoring the different jumps, could be arguably linear, the presence of this (large) negative values suggest the heavy impact of numerical errors for what concerns the computation of the stepsize.

If we look at Fig. 3.10 we can observe how line search converges compared to the optimal stepsize in relatively well conditioned problems. In this cases it is clear that exact line search is converging to the global minimum, having a relative gap in the order of 10^{-14} or 10^{-15}.

Execution time analysis for different stepsizes

When discussing the potential benefits in using exact line search, we remarked that the main trade-off present when adopting it is that we need to pay more computational cost in order to find the optimal stepsize at each iteration. This is exactly what is shown in Fig. 3.11a, where

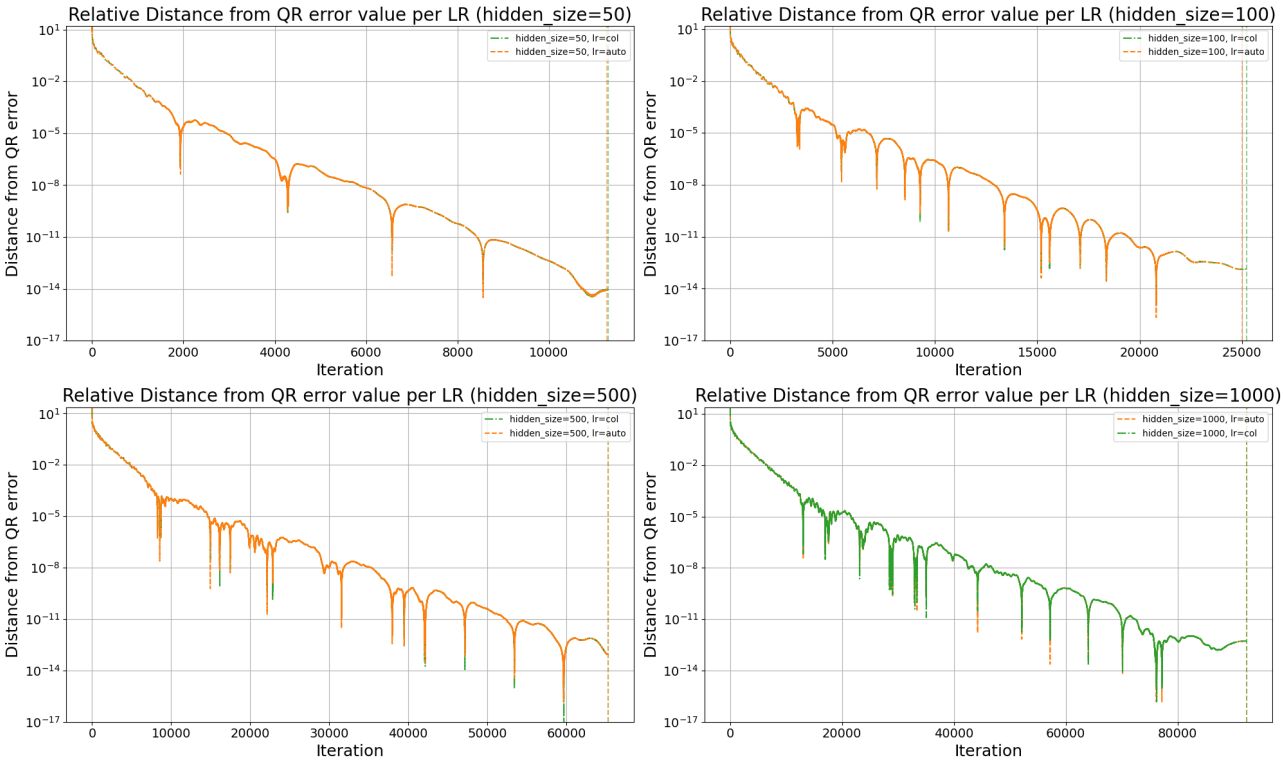


Figure 3.9: Relative difference of the LSE of line search with optimal β and LSE of QR factorization.

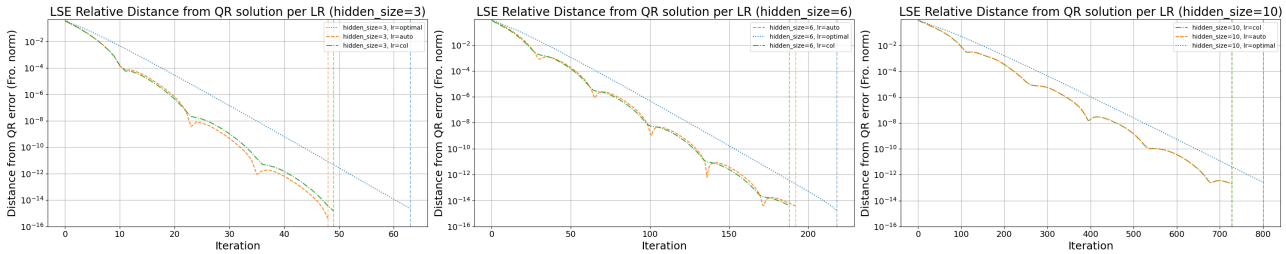
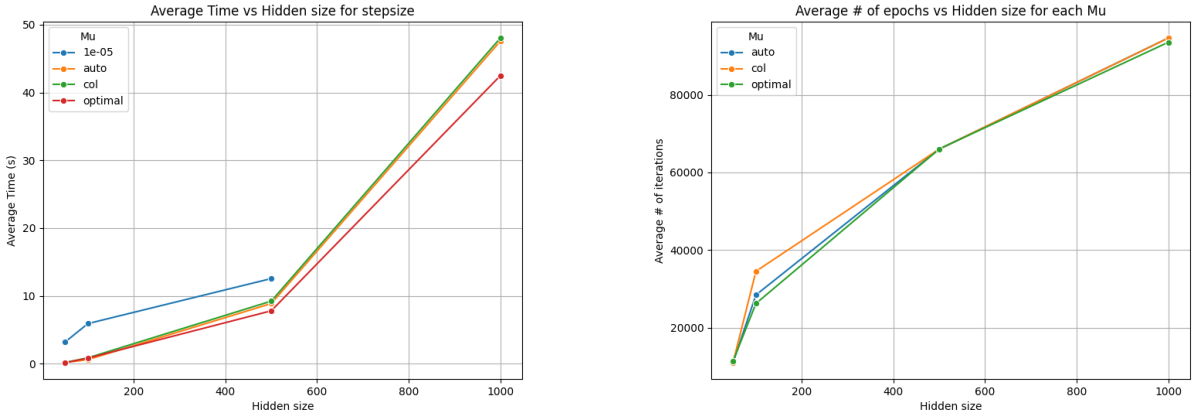


Figure 3.10: Relative difference in function value w.r.t. QR decomposition for small models. All runs were executed with $\alpha = 10^{-2}$, optimal β. $\epsilon = 10^{-10}$ for 10 units, while $\epsilon = 10^{-11}$ for 3 and 6 units models.



(a) (b)

Figure 3.11: (a) Execution time of NAG(b) Amount of iterations needed for convergence of NAG. All datapoints are the average over 3 runs, obtained by executing 5 runs and removing the one with highest and lowest execution time. All runs are executed with $\alpha = 10^{-2}$

both methods of line search employed, while following the same trend as the optimal stepsize, are always slower. This is explained by looking at Fig. 3.11b, were it is immediately evident that in practice, as long as we are able identify the optimal values of μ and β, we can achieve convergence in an amount of iterations comparable to the line search.

However, it is important to note that in scenarios where determining the optimal step size is computationally unfeasible, line search often yields faster results than arbitrarily chosen step sizes, such as 10^{-5}. Of course, this advantage must be considered together with the observed lack of reliable convergence exhibited by line search across the examples previously presented. As such, the potential reduction in execution time offered by line search is significantly undermined by this drawback, limiting its practical applicability in large models.

As a final consideration, the execution time shown in Fig. 3.11a is not exactly quadratic, as when increasing the model size we can observe a steeper increase. This is caused by the related increase in the amount of iterations, which is related to the condition number of the problem.

Convergence analysis for different β

In the previous chapter, we demonstrated that, under optimal step size and momentum hyperparameters, the convergence speed of the algorithm could improve from $O(\kappa * log(\frac{1}{\epsilon}))$ to $O(\sqrt{\kappa} log(\frac{1}{\epsilon}))$. This implies that incorporating momentum should enhance the convergence speed by a factor of $\sqrt{\kappa}$. To assess how this theoretical improvement we start by examining Fig. 3.12, which presents the gradient norm for different values of β. In these experiments, we employ optimal μ (as per Eq. 2.1) and $\alpha = 10^{-2}$ for consistency with previous experiments.

Let us start from the most evident fact in the plots, that is the behavior of the scheduling approach we previously mentioned in Chapter 2. Indeed, when using the scheduling we observe a very irregular behavior in the gradient, with what seems to be relatively sublinear convergence, although for larger model sizes the algorithm seems to be unable to converge at all.

For the fixed values of β instead we observe the expected behavior, that is the gradient norm is converging linearly until reaching ϵ, and the closer β is to the optimal value the faster the convergence is.

This behavior is further confirmed by Fig. 3.13 and Fig. 3.14, which show respectively the distance from the QR solution and the difference in function value. While Fig. 3.13 still shows a similar linear trend, Fig. 3.14, apart from confirming that as long as we are able to choose a

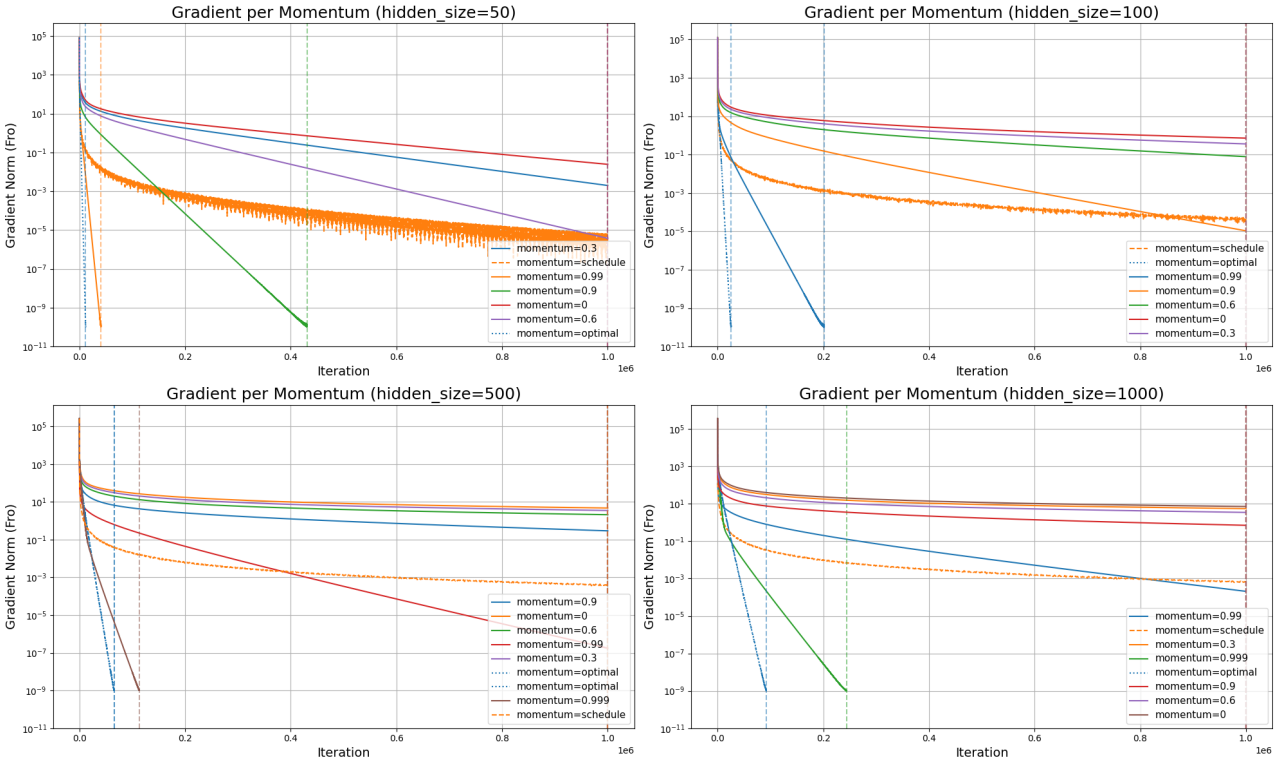


Figure 3.12: Gradient convergence for different values of β and different model sizes.

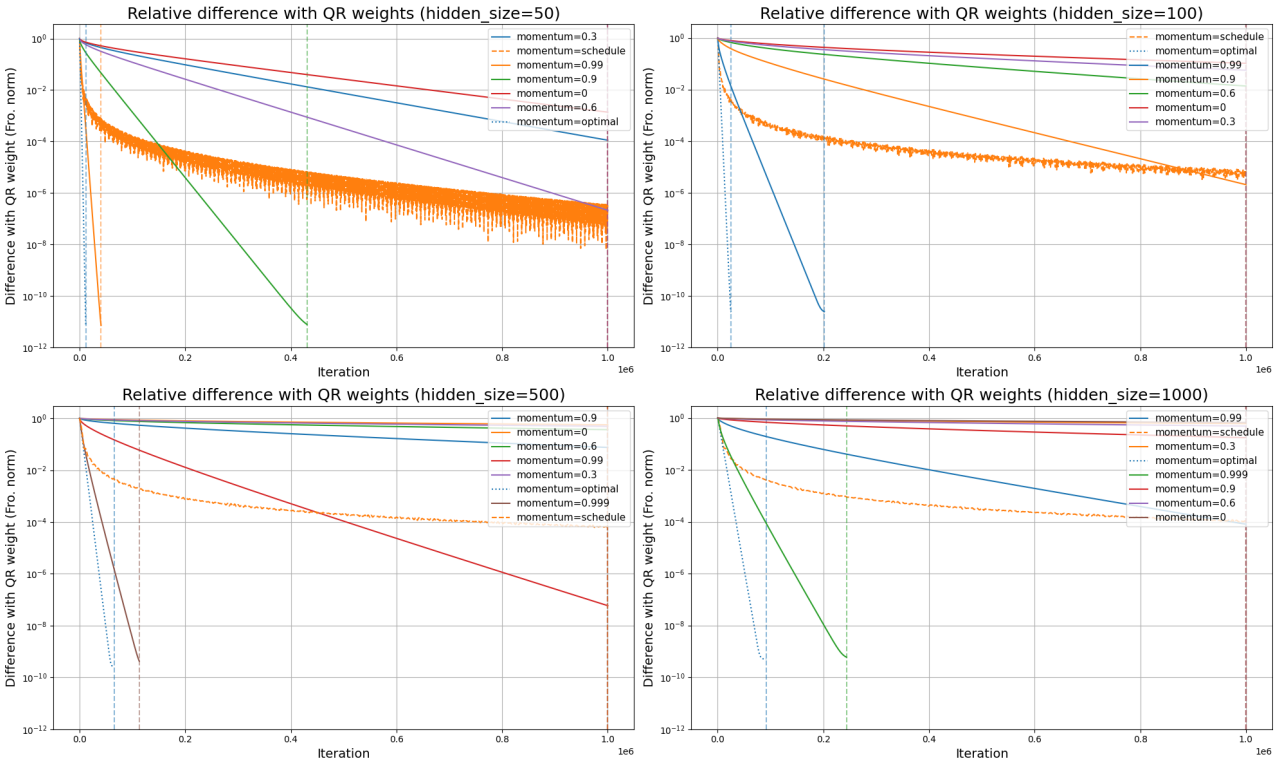


Figure 3.13: Relative distance from QR solution of NAG for different sizes and β.

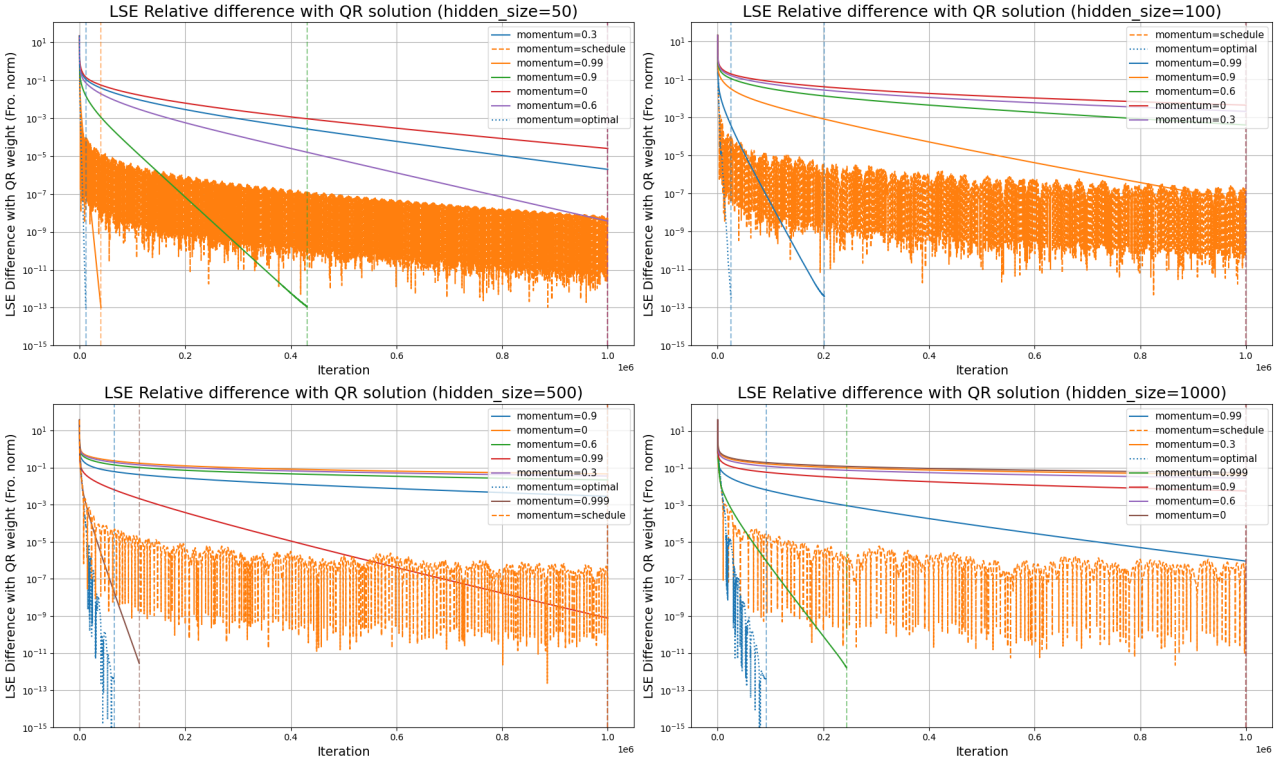


Figure 3.14: Relative difference between QR solution LSE and NAG solution LSE for different values of β.

reasonable β value we can converge linearly to a good solution, it also shows that our scheduling approach fails to converge.

To better understand what is happening when using our scheduling, let us look at Fig. 3.15, showing the difference between the scheduling function value and the QR solution function value in a symlog scale. The algorithm in this case seems to gradually converge to the QR value, although this oscillating behavior, characterized also by very large negative differences in early iterations, is quite peculiar. In fact, especially in the initial iterations, there are negative differences in the order of $\sim 10^{-5}$.

In fact, this behavior is not specifically a cause of the ill-conditioning of the B^TB matrix, but it is exactly a problem that characterizes the scheduling approach itself. If we look at Fig. 3.16 we still needed to report the absolute differences in function value, meaning that while the plots look relatively nicer, the approach still has problems regarding its convergence.

The impact of the regularization parameter

The last analysis we propose aims to inspect how changing the value of α impacts the speed of convergence of the algorithm, whose results are shown in Fig. 3.17. It is important to remark the difference in impact that changing α has for different model sizes: while for smaller size (50 and 100 units) the relative decrease in the amount of iterations is not substantial, there is a significant reduction of iterations for larger models.

The cause of this behavior is related to the conditioning of the A^TA matrix, and in particular its eigenvalues. In fact in all of our runs, the larger models always had small negative eigenvalues, in the order of $\sim -10^{-13}$. When computing B^TB we registered and higher improvement in the conditioning in larger model, as also shown previously in Fig. 3.2, where for 100 units A^TA and B^TB have a conditioning of $\sim 10^6$ (using $\alpha = 10^{-2}$) while for 1000 hidden units the condition number went from 10^{18} of A^TA to 10^7 in B^TB.

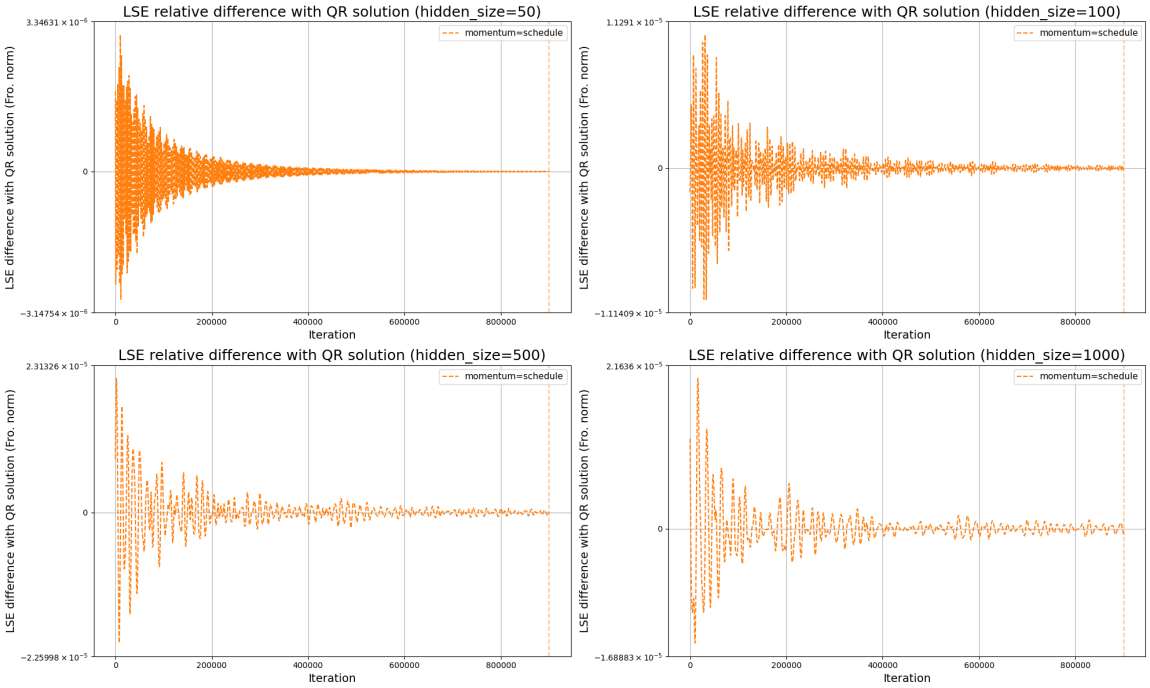


Figure 3.15: Relative function value difference from QR solution for scheduling approach

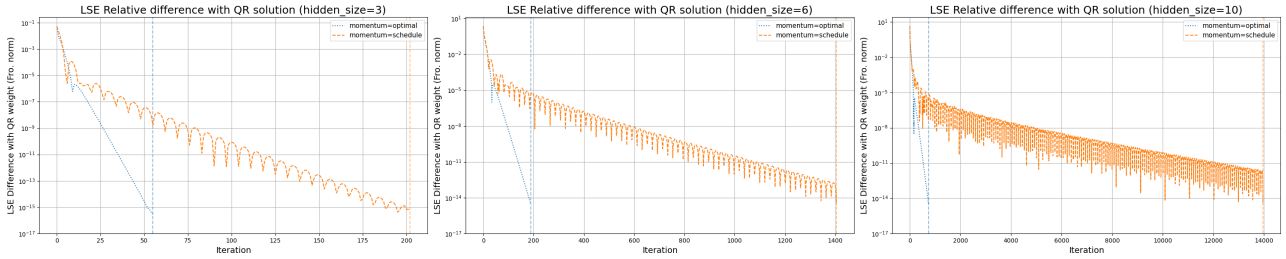


Figure 3.16: Scheduling vs optimal β in small models. We report the relative differences with absolute values, in fact the scheduling obtains function values lower than the one obtained from QR decomposition.

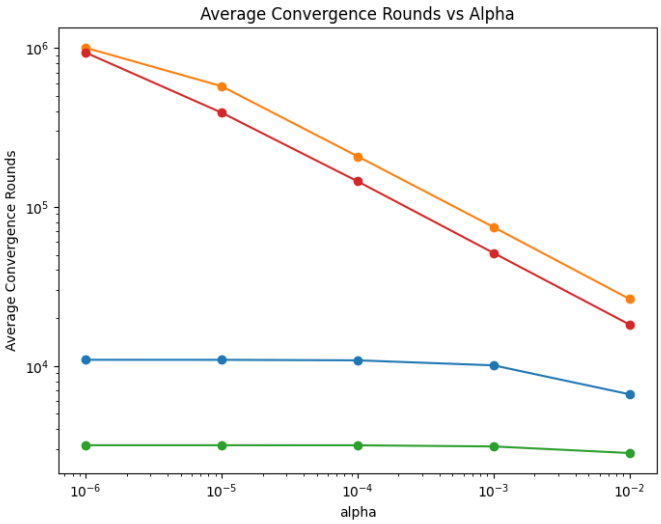


Figure 3.17: Average amount of iterations needed for the model sizes inspected when changing α. All measurements are the mean over 4 runs, executed with optimal μ and β.

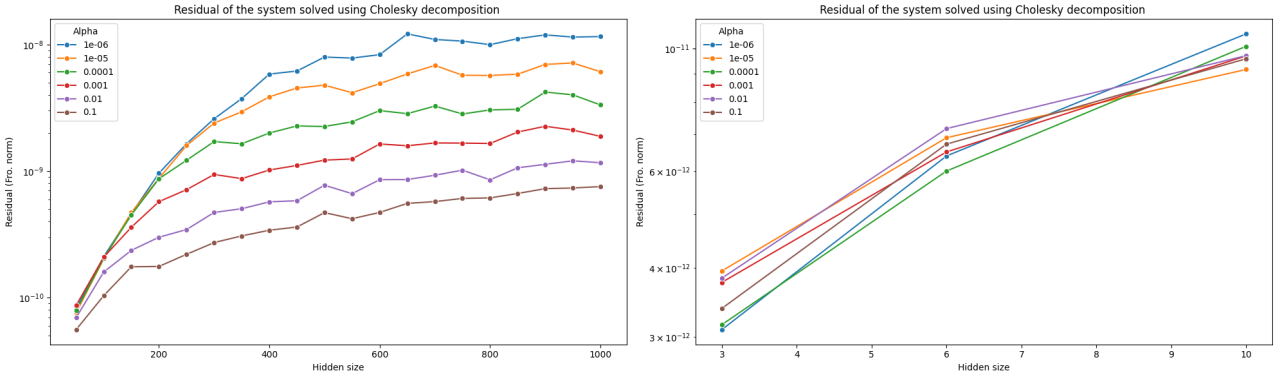


Figure 3.18: Residual of the system solved using Cholesky decomposition. Results are the average over 5 runs.

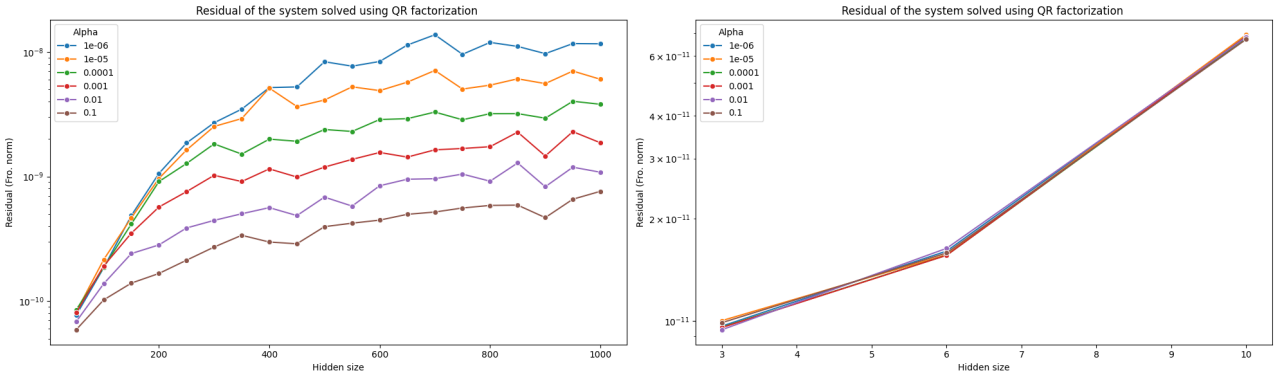


Figure 3.19: Residual of the system solved using QR factorization. Results are the average over 5 runs.

3.3.2 Experiments on the closed formula solution

When inspecting our implementation of the closed-formula solution for solving the Extreme Learning problem we are mainly interested in analyzing:

- The quality of the solution obtained;
- The execution time of our implementation;
- The empirical impact of α on the solution found by the algorithm.

Considerations on the residual of the Cholesky decomposition

Before investigating the quality of the solution of our Least Square problem (LSE) we find it appropriate to discuss momentarily the residual obtained from the resolution of our system using the Cholesky decomposition. Let us first look at Fig. 3.18, showing the residual obtained for different model sizes and values of α.

Ideally, when solving a system of this kind, one would like to obtain a very low residual, possibly in the order of machine precision. While this may be the case for small models with 10 or less units, this is clearly not true when increasing the model size, as the residual norm usually ranges from 10^{-10} to 10^{-8}, depending on the value of α.

As a reference, we also show the residual obtained by solving the system using QR factorization in Fig. 3.19. What we observe is a similar behavior to the one shown in Fig. 3.18. This is expected, as both procedures are backward stable. In fact, the primary limitation of

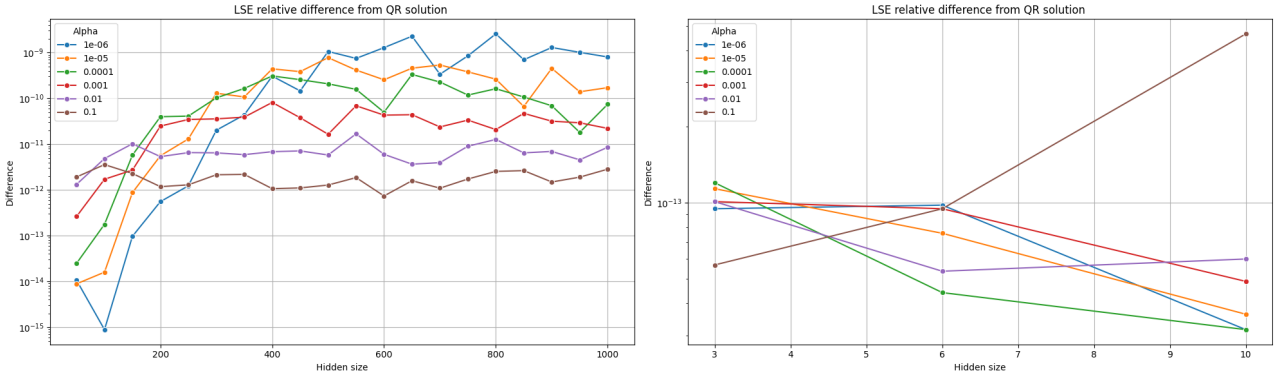


Figure 3.20: Relative difference between LSE of Cholesky decomposition solution and QR factorization solution

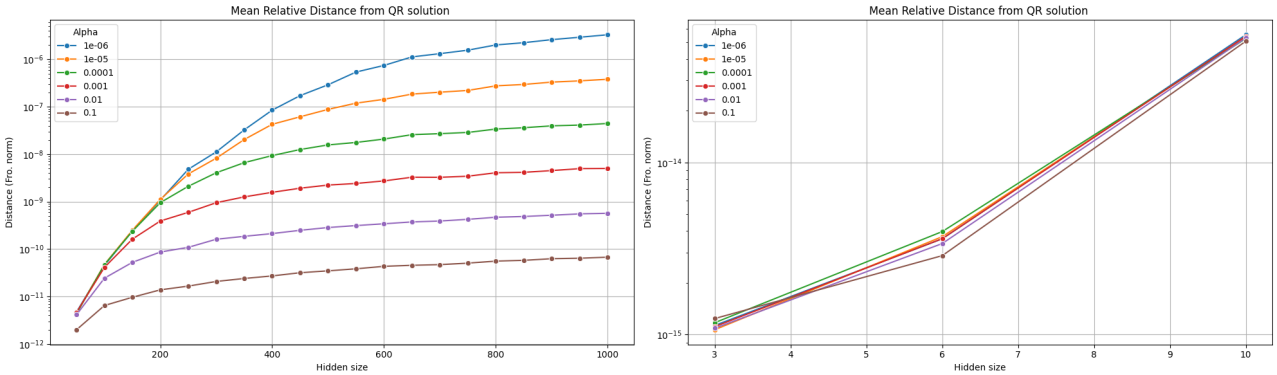


Figure 3.21: Relative distance from QR factorization solution in Frobenius norm, over 5 experiments.

the closed-form solution based on Cholesky decomposition does not originate from the decomposition itself, but rather from the matrix product A^TA, which is discussed in the following section.

Considerations on solution quality

To asses the solution quality of our closed-formula algorithm we compare it against the solution found by using QR decomposition, which is expected to be closer to the true function minima due to the backward stability of the QR decomposition.

Let us first look into Fig. 3.20, which shows the distance between the two solutions. From the plots, we immediately observe that we never registered an experiment where the Cholesky decomposition closed-formula managed to find a better solution than QR factorization. This is in agreement with what the theory tells us, as indeed the closed-formula using the pseudoinverse is expected to yield a less precise solution when compared to QR factorization due to not being backward stable.

It is also interesting to notice the trend shown for different α when changing the model size. In fact, if for $\alpha = 10^{-1}$ the difference between the two errors remains the same across different model sizes, we can observe that for lower values of α in smaller models we register a difference that is in the order of 10^{-15}, and this difference increases up to 10^{-9} for larger model sizes. This behavior can be explained by observing the increase in the condition number we previously shown in Fig. 3.2. For smaller, well conditioned matrices, our closed formula is able to obtain a solution relatively closer to the true minima, while for larger matrices the quality of the solution obtained deteriorates.

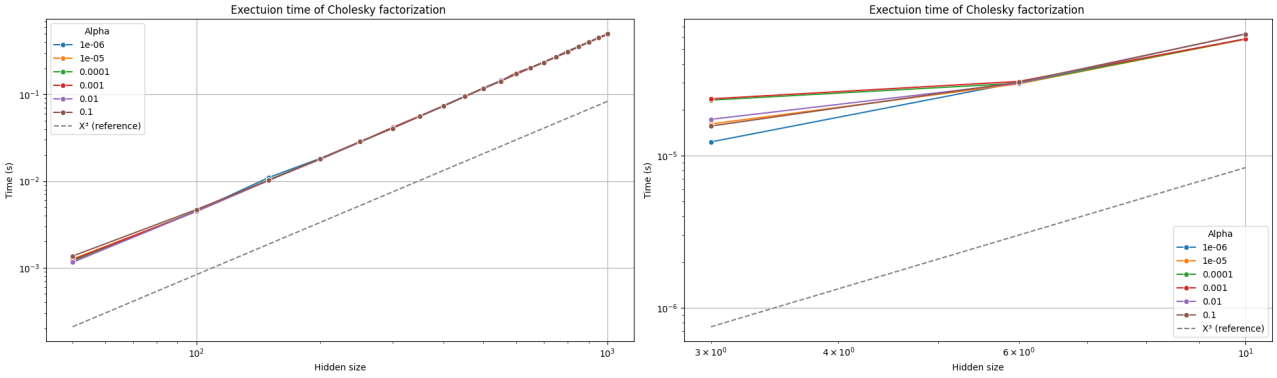


Figure 3.22: Execution time of our implementation of the cholesky decomposition

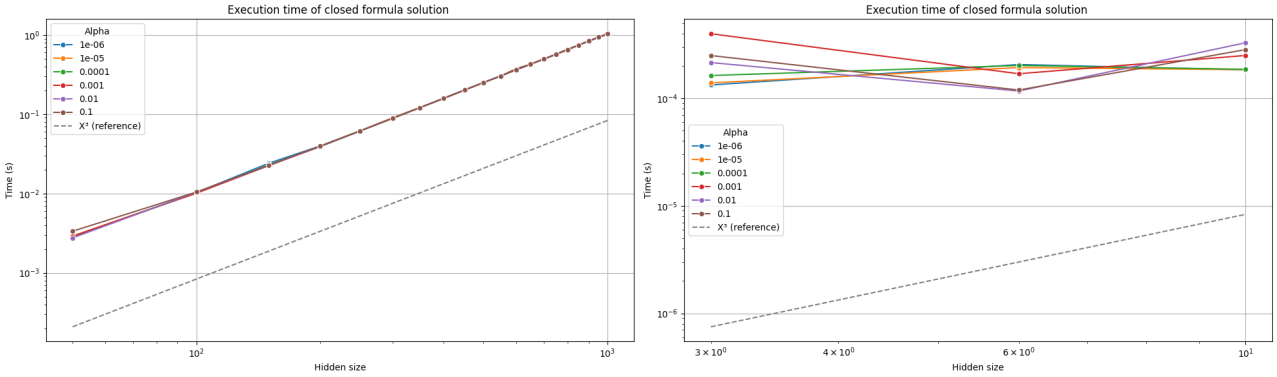


Figure 3.23: Total execution time of our closed formula solution

In fact, let us look at Fig. 3.21, showing the distance between the two solutions. What we can indeed see is the behavior we just anticipated. The smaller α and the model are, the closer the solutions are, due to better conditioning, while when we start increasing either the model size or the value of α we observe large differences in the two, up to 10^{-2} when using $\alpha = 10^{-6}$.

Considerations on execution time

The last considerations we make about our closed formula solution are about the execution time. Fig. 3.22 and Fig. 3.23 show the execution time of the Cholesky decomposition and the execution time of the closed-formula algorithm, respectively. Both plots exhibit the same, cubic trend, as proved by the reference cubic curve present in the plots.

It is particularly important to notice that, while in theory for $h < L$ the matrix product should be the more costly operation, in the order of $O(L^2h)$, in practice, since the product is computed using library functions, its computation time is neglegible, especially when compared to the execution time of our implementation of Cholesky decomposition.

3.3.3 Comparison Between the Closed-Form Solution and NAG

In the final analysis presented in this report, we compare the closed-form solution obtained via Cholesky decomposition with the NAG algorithm, considering both solution accuracy and computational efficiency. A summary of the results is provided in Table 3.1.

The table clearly illustrates that, for $\alpha = 10^{-2}$, the closed-form method is not only significantly faster than NAG, but also yields a solution that is closer to the true minimum, with improvements evident from approximately the 11^{th} decimal place onward.

While the higher precision of the closed-form solution is expected—since its accuracy is not

Hidden size	LSE (Chol)	LSE (NAG)	Total time Chol. (s)	Total time NAG (s)
50	51.600718006360	51.600718006365	0.0027	0.1541
100	33.316243589972	33.316243589975	0.0103	4.703
500	24.46055821017	24.4605582102**2**	0.2510	9.3887
1000	23.78054064414	23.7805406441**8**	1.0301	52.9902

Table 3.1: Solution quality and total execution time for the two methods discussed in the report. The higher (worse) Least square error is marked in **bold**, from the decimal it is worse. The measurements are the average among 4 seeds for $\alpha = 10^{-2}$. Execution times for NAG were taken using the optimal configuration of μ and β, and the LSE is the one obtained from the fastest fixed step for each model (as the optimal parameters could yield unstable solutions).

dependent on the manually tuned parameter ϵ—the substantial gain in computational speed relative to the gradient-based approach was not anticipated.

It is worth noting that the execution time of NAG was measured during a second run of the algorithm, in which all logging operations were disabled. This was done to ensure a fairer comparison with the closed-form method. In practice, if logging were enabled, the execution time would be considerably higher, potentially by an order of magnitude.

Chapter 4

Conclusions

In this work, we explored two primary approaches for training Extreme Learning Machines. Initially, we defined the learning problem within the ELM framework, presenting it as a least-squares optimization problem. Then, we introduced the two main approaches discussed throughout the report: first, Nesterov Accelerated Gradient (NAG) an iterative method within the accelerated gradient descent family, and second, a direct closed-form solution derived via the Moore-Penrose pseudoinverse utilizing Cholesky decomposition.

We then proceeded to analyze the performance of these two algorithms on our specific problem. Through experiments, we confirmed the expected behaviors, which aligned well with our theoretical predictions. Our comparison between the two algorithms highlighted a significant distinction: although theoretically comparable, the closed-form solution demonstrated a significant advantage both in terms of speed and solution quality over the iterative NAG approach.

In order to enhance this study, an area worth exploring is a more extensive examination of model size variations. Although we conducted some preliminary tests on different model sizes, increasing the dimensionality further could provide valuable insights. However, it is important to note that increasing model size may amplify computational limitations in NAG, particularly when compared to the more efficient closed-form approach.

In conclusion, our investigation confirms that both training approaches for ELMs hold theoretical and practical validity, with the closed-form solution emerging as the more efficient and accurate option in practice.

Bibliography

[1] Sébastien Bubeck. *Convex Optimization: Algorithms and Complexity*. 2015. arXiv: 1405.4980 [math.OC]. URL: https://arxiv.org/abs/1405.4980.

[2] Tom Goldstein, Christoph Studer, and Richard Baraniuk. *A Field Guide to Forward-Backward Splitting with a FASTA Implementation*. 2016. arXiv: 1411.3406 [cs.NA]. URL: https://arxiv.org/abs/1411.3406.

[3] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[4] Yuege Xie, Xiaoxia Wu, and Rachel Ward. *Linear Convergence of Adaptive Stochastic Gradient Descent*. 2020. arXiv: 1908.10525 [stat.ML]. URL: https://arxiv.org/abs/1908.10525.

Appendix A

Appendix

A.1 Proof bound of condition number for rectangular matrices.

Let $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$. We want to prove whether:

$$\text{cond}(AB) = \frac{\sigma_1(AB)}{\sigma_{\min(m,p)}(AB)} \overset{?}{\leq} \frac{\sigma_1(A)}{\sigma_{\min(m,n)}(A)} \cdot \frac{\sigma_1(B)}{\sigma_{\min(n,p)}(B)} = \text{cond}(A)\cdot\text{cond}(B).$$

$\sigma_1(AB) \leq \sigma_1(A)\sigma_1(B)$ always holds; the question is whether

$$\sigma_{\min(m,p)}(AB) \overset{?}{\geq} \sigma_{\min(m,n)}(A)\cdot\sigma_{\min(n,p)}(B).$$

If we assume that $m \geq n \geq p$, it holds. This is because:

- If $\sigma_p(B) = 0$ the inequality holds trivially.
- In general note that, if $C \in \mathbb{R}^{q\times r}$ with $q \geq r$, then

$$\sigma_r(C) = \sqrt{\lambda_{\min}(C^TC)} = \sqrt{\inf_{x\in\mathbb{R}^r_*} \frac{x^TC^TCx}{x^Tx}} = \inf_{x\in\mathbb{R}^r_*} \frac{\|Cx\|}{\|x\|}$$

  where $\mathbb{R}^r_* = \mathbb{R}^r \setminus \{0\}$.

This means that, because $Bx \neq 0$ for $x \neq 0$, we can observe the following:

$$\begin{aligned}
\sigma_p(AB) &= \inf_{x\in\mathbb{R}^p_*} \frac{\|ABx\|}{\|x\|} \\
&= \inf_{x\in\mathbb{R}^p_*} \frac{\|ABx\|\cdot\|Bx\|}{\|Bx\|\cdot\|x\|} \\
&\geq \left(\inf_{x\in\mathbb{R}^p_*} \frac{\|ABx\|}{\|Bx\|}\right)\left(\inf_{x\in\mathbb{R}^p_*} \frac{\|Bx\|}{\|x\|}\right) \\
&\geq \left(\inf_{y\in\mathbb{R}^n_*} \frac{\|Ay\|}{\|y\|}\right)\left(\inf_{x\in\mathbb{R}^p_*} \frac{\|Bx\|}{\|x\|}\right) \\
&= \sigma_n(A)\cdot\sigma_p(B).
\end{aligned}$$

And we have that the condition number of the matrix AB is bounded by the condition number of A and B. The proof also holds for $m \leq n \leq p$ by transposing everything.