Computational Mathematics for Learning and Data Analysis - Project ML-3

Davide Montagno B., Alex Pasquali

Contents

1 Quick overview

This project consists of an implementation from scratch of two models, a neural network with L1 regularization and a L2 linear regression (namely M1 and M2), on which to apply the following 3 algorithms:

A1) A standard momentum approach applied to M1. Our choice is the *Classical Momentum / Heavy Ball method* by *Polyak*.

A2) An accelerated gradient method applied to M1. The choice is the *Nesterov's Accelerated Gradient*.

A3) A basic version of the direct linear least squares solver applied to M2.

2 M1: neural network with L1 regularization

The model on which to apply both A1 and A2 is a fully-connected feed-forward neural network (NN) composed by: one input layer, hidden layers (one or more) and finally one output layer.
Next we introduce some concepts that will be useful to understand the NN's behaviour, its construction and the general properties. Later on, during the experiments, we will describe its performance.

2.1 Activation Function

Each unit of the NN performs the following operation:

$$h_j = \sigma(\sum_{i=1}^{n} w_{ji} \cdot h_i + b_j) \tag{1}$$

where h_j is the current unit's output, h_i are the outputs of the previous layer (i.e. the current unit's inputs), b_j is the bias and w_{ji} is the weight connecting unit i to unit j. σ is the **activation function**, that takes the sum and produces the unit's output.
Equivalently, in a matrix form, each layer performs:

$$\mathbf{h}_l = \sigma(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l) \tag{2}$$

where l indicates the layer and $\mathbf{h}_l$ is now a vector containing the outputs of all the units in the l-th layer.
The activation function has to respect the following properties:

- **Non-linearity**: a feed-forward NN with (at least) one hidden layer equipped with a continuous non-linear activation function can approximate arbitrarily well any given function [1]. Moreover, this property is used by the NN to perform an adaptive basis expansion to create the most suitable internal representation of the data (the one that best simplifies the job of the output layer).

- **Differentiability**: activation functions should be differentiable since the learning algorithm is based on the gradient of the loss function, that involves the derivative of such activations.
- **Finite range**: if the output range of the activation functions is finite and reasonably small, the learning algorithm tends to be more stable.

Our choice is the *Logistic function* (also referred to as *Sigmoidal/Sigmoid* in the scope of this report), constructed as: $\sigma(x) = \frac{1}{1+e^{-z}}$.
In Figure 1 we can see its shape.

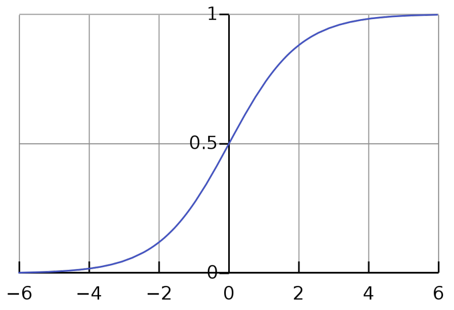

Figure 1: Logistic Activation Function

2.1.1 Output activation function

For the activation function of the output layer the situation is a little different: depending on the task, a different output range may be needed. Therefore, for binary classification, a suitable activation function would still be the *sigmoidal function*, that compresses the output in the range [0, 1] and tends to saturate (assume values near to the extremes of its range), polarizing towards a class or the other.
Concerning a regression task, a *linear* activation ($f(x) = x$) is needed to have a linear, non-biased response of the output layer. In this case, saturation is to be avoided, especially if the range of the target variables exceeds the one of the activation function.

2.2 Objective Function

Usually, the main goal for a NN is to achieve a low error and a good generalization. This can be obtained using a loss function plus a regularization, making our objective function in the form of:

$$J(\mathbf{W}) = \mathcal{L}(\mathbf{W}) + \lambda\Omega(\mathbf{W}) \tag{3}$$

where $\mathcal{L}(\mathbf{W})$ is the loss function, $\Omega(\mathbf{W})$ is the regularization and λ is the regularization coefficient (a hyper-parameter).

In literature there are many types of loss functions, for our project we decided to use the well known *Mean Square Error* (MSE) for all the tasks we are dealing with.

2.2.1 Mean Square Error (MSE)

In this section we will analyze in details the *Mean Square Error* function starting from its formula:

$$\text{MSE} = \frac{1}{2m}\sum_{i=1}^{m}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{4}$$

where m represents the number of samples for which we computed the outputs $\hat{\mathbf{y}}_i$ before proceeding with the weight update. We can represent this functions also as a composition of the square function and the Euclidean norm by writing the previous formula as:

$$\text{MSE} = \frac{1}{2m}\|\mathbf{Y} - \hat{\mathbf{Y}}\|_2^2 \tag{5}$$

where $\hat{\mathbf{Y}}$ is a vector or matrix containing the net's outputs for all the m input samples. Since this formula is used to calculate the error of our models, our primary goal is to minimize it, and we do so by setting the gradient to zero (more details on this in the following sections). Of course, in order to make this operation possible, it should be a *differentiable* function. The gradient for the MSE with respect to $\hat{\mathbf{Y}}$ is defined as:

$$\nabla_{\hat{\mathbf{Y}}}\text{MSE}(\hat{\mathbf{Y}}, \mathbf{Y}) = \frac{1}{m}(\hat{\mathbf{Y}} - \mathbf{Y}) \ . \tag{6}$$

Now let us quickly recap the relevant properties of the MSE.

2.2.2 Properties of MSE

2.2.2.1 Continuity By definition, a function f is L-Lipschitz continuous if there exists a constant L such that:

$$\begin{gathered}\|f(\mathbf{x}_1, \mathbf{y}_1) - f(\mathbf{x}_2, \mathbf{y}_2)\| \leq L(\|\mathbf{x}_1 - \mathbf{x}_2\| + \|\mathbf{y}_1 - \mathbf{y}_2\|) \\ \forall(\mathbf{x}_1, \mathbf{y_1}), (\mathbf{x}_2,\mathbf{y_2}) \ \in \ \textbf{dom}\ f.\end{gathered} \tag{7}$$

Since the MSE is a composition of the Euclidean norm and the square function, it is L-Lipschitz continuous only if we restrict its domain to a bounded set. In Section 2.1, we said that each layer of the NN has a sigmoid activation function, which is a squashing function that limits every layer's output to the range $(0, 1)$, therefore, the MSE applied to our network can be considered a **continuous** function.

2.2.2.2 Differentiability

The sigmoid function is continuously differentiable with bounded Lipschitz continuous derivative. Thanks to this observation, we can consider the MSE a **differentiable** function, since it is made of a composition of differentiable functions (namely, the sigmoid functions that compose the network).

2.2.2.3 Convexity

Finally, the squared Euclidean distance is convex: as any distance, it is non-negative and the square function is convex and increasing for non-negative values.
It may happen, anyway, that the function in question is not convex if it is composed of other types of functions. By definition, for $h : \mathbb{R}^k \to \mathbb{R}$, $g_i : \mathbb{R}^n \to \mathbb{R}$, the function

$$f(\mathbf{x}) = h(g_1(\mathbf{x}), \dots, g_k(\mathbf{x}))$$

is convex when h is non-decreasing in each argument and g_i are convex. Since the functions that compose our networks, i.e. the sigmoid functions, are not convex, in our case **we cannot rely on the MSE being convex**, thus we will not use that assumption in Section 3.

2.2.3 L1 regularization

Getting the ojective function too low may harm the generalization capability of the model (overfitting). To have a tradeoff between the two, it is possible to introduce some regularization by adding a penalty term to the loss itself ($\mathcal{L}(\mathbf{W}) + \lambda\Omega(\mathbf{W})$).
In the case of L1 regularization (a.k.a. Lasso), $\Omega(\mathbf{W}) = \|\mathbf{W}\|_1$. This tends to make some weights to be **exactly** 0, making $\mathbf{W}$ sparse and $J(\mathbf{W})$ non-differentiable. In the case of digital floating point computations, it is unlikely to get a "perfect" 0, so we might proceed assuming that this never happens, thus treating the objective function as differentiable. Anyway, we can always set $\frac{\partial \|w_{ij}\|_1}{\partial w_{ij}} = 0$ if $w_{ij} = 0$, therefore getting a sub-gradient of $J(\mathbf{W})$, so, even if some weights are actually 0, this does not represent a problem, as shown in Section 3.

2.3 Solving the learning problem

After having explained our goal and the properties of our objective function, in this section we explain how to minimize it in the context of neural networks, by adjusting the weights of the net itself.
The classic approach in machine learning is to use a gradient-based optimization technique. Let us recall that the gradient of a function is the vector of all its partial derivatives and it points towards the direction of maximum growth of the function itself.
Gradient Descent (GD) is a first-order iterative algorithm that, at each iteration, computes the following update:

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \eta \nabla_{\mathbf{W}_k} J(\mathbf{W}_k) \tag{8}$$

where k is the iteration index, $\mathbf{W}$ indicates the weights, $\nabla_{\mathbf{W}_k}$ indicates the gradient w.r.t. the weights and $J(\mathbf{W})$ is the objective function (loss + regularization). Finally, η is the *learning rate* or *stepsize*: a hyper-parameter that regulates the magnitute of the update.
From a practical point of view in the context of NNs, at each iteration of GD, two operations are performed: *forward-propagation*, to compute $\mathcal{L}(\mathbf{W})$, and *backward-propagation*, to efficiently compute $\nabla J(\mathbf{W})$. This algorithm is usually referred to as *backpropagation* or *backprop* [2].

2.3.1 Forward-propagation

The network is fed at each time with one input $\mathbf{x}_k$ and computes the corresponding output $\hat{\mathbf{y}}_k$. Then $\mathcal{L}(\hat{\mathbf{y}}_k, \mathbf{y}_k)$ represents the loss for the k-th pattern. Pseudo-code in Algorithm 1.

Algorithm 1 Forward propagation through a NN.
$\mathbf{x}$ is the current input pattern, L is the number of layers.
Here the loss function is MSE, but it may as well be a different one.

```
1: procedure FORWARD PROPAGATION($\mathbf{x}$)
2:     $\mathbf{h}_0 \leftarrow \mathbf{x}$
3:     for $l = 1, \dots, L$ do
4:         $\mathbf{a}_l \leftarrow \mathbf{W}_l \cdot \mathbf{h}_{l-1} + \mathbf{b}_l$
5:         $\mathbf{h}_l \leftarrow \sigma(\mathbf{a}_l)$
6:     $\hat{\mathbf{y}} \leftarrow \mathbf{h}_L$
7:     $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \mathrm{MSE}(\hat{\mathbf{y}}, \mathbf{y})$
8:     $J(\mathbf{W}, \hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda\Omega(\mathbf{W})$
```

2.3.2 Backward-propagation

For the second step we must compute the gradient of the objective function w.r.t. the parameters, namely $\nabla_{\mathbf{W}} J(\mathbf{W}, \hat{\mathbf{y}}, \mathbf{y})$. We must remember that each layer of our architecture has its own activation function, that is applied to the output of each neuron. This phase allows the error information to flow backward through the layers in order to compute the gradient by applying the *Chain Rule of Calculus* (used for computing the derivative of a composition of functions). Pseudo-code in Algorithm 2.

Algorithm 2 Backward propagation of the error signal through a NN.
L is the number of layers. $\mathbf{a}_l$ and $\mathbf{h}_l$ have the same meaning they have in algorithm 1. The $\odot$ symbol represents the element-wise product. The subscript after the ∇ symbol indicates that the gradient is computed w.r.t. the subscript itself (e.g. $\nabla_t f(t)$ means "the gradient of f w.r.t. t"). We use the abbreviation $\nabla_{sub} J$ in place of $\nabla_{sub} J(\mathbf{W}, \hat{\mathbf{y}}, \mathbf{y})$.

```
1: procedure Backward propagation
2:     $\delta_L \leftarrow \nabla_{\hat{\mathbf{y}}} J \;=\; \nabla_{\hat{\mathbf{y}}} \mathcal{L} + \lambda \nabla_{\hat{\mathbf{y}}} \Omega(\mathbf{W}) \;=\; \nabla_{\hat{\mathbf{y}}} \mathcal{L} + 0 \;=\; \hat{\mathbf{y}} - \mathbf{y}$
3:     for $l \;=\; L,\; L-1,\; \ldots,\; 1$ do
4:         $\delta_l \leftarrow \nabla_{\mathbf{a}_l} J \;=\; \delta_l \odot \nabla_{\mathbf{a}_l} \mathbf{h}_l \;=\; \delta_l \odot \sigma'(\mathbf{a}_l)$
5:         $\nabla_{\mathbf{b}_l} \mathcal{L} = \delta_l$
6:         $\nabla_{\mathbf{W}_l} \mathcal{L} = \delta_l \cdot \mathbf{h}_{l-1}^T$
7:         $\nabla_{\mathbf{b}_l} J = \nabla_{\mathbf{b}_l} \mathcal{L} + \lambda \nabla_{\mathbf{b}_l} \Omega(\mathbf{W})$
8:         $\nabla_{\mathbf{W}_l} J = \nabla_{\mathbf{W}_l} \mathcal{L} + \lambda \nabla_{\mathbf{W}_l} \Omega(\mathbf{W})$
9:         $\delta_{l-1} \leftarrow \delta_l^T \mathbf{W}_l$
```

In lines 7 and 8 of Algorithm 2 there is the gradient of the regularization term w.r.t. the biases and weights. For every parameter (i.e. weight or bias) w_{ij}, $\lambda \nabla_{w_{ij}} \Omega(\mathbf{W}) = \lambda |w_{ij}|$ in case of L1 regularization.

3 Optimization algorithms

In this section, we will discuss the optimization algorithms of choice, their properties and what we can expect as results.
First we briefly present the *Stochastic Gradient Descent* algorithm (SGD), which is the base for the next algorithms we are going to describe.

3.1 Stochastic Gradient Descent (SGD)

SGD can be seen as a variation of GD where weight updates are performed using an approximation of the gradient computed on a *minibatch* of data, i.e. a random subsample of the training set. In fact, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data-generating distribution [3]. We will call this approximated gradient $\mathcal{G}(\mathbf{W}_k, \xi_k)$, where ξ_k is a random variable expressing the noise caused by the approximation, that is subject to the property: $\mathbb{E}_{\xi_k}[\mathcal{G}(\mathbf{W}_k, \xi_k)] = \nabla J(\mathbf{W}_k)$. SGD makes the computation of a single update much more efficient, but less accurate: the smaller the size of the minibatches, the stronger the noise introduced by the approximation. This leads to the well known zigzag behavior illustrated in Figure 2.

As stated in [3, 4], classic convergence analysis of the SGD algorithm for non-convex smooth functions relies on conditions on the stepsizes η_k. A sufficient condition for SGD convergence is that the learning rate is positive and decreases

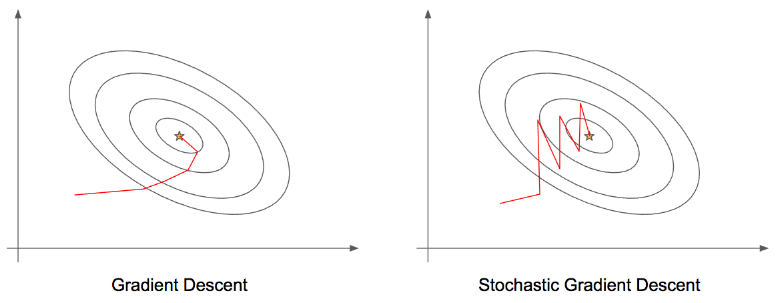

Figure 2: GD vs. SGD behavior (illustrative purpose, this picture does not come from our experiments)

over time, thus satisfies the following:

$$\sum_{k=1}^{\infty} \eta_k = \infty, \qquad \sum_{k=1}^{\infty} \eta_k^2 < \infty \ . \tag{9}$$

3.2 Momentum

Momentum is a first-order acceleration technique for gradient-based algorithms (such as SGD) that takes into consideration the past estimates of the (sub)-gradient while performing a weight update in order to have a faster convergence.
In the scope of this project, we have to implement a standard momentum descent approach (A1) and an algorithm of the class of accelerated gradient methods (A2). For the first one we chose the *Classical Momentum/Heavy Ball*, while for the second one our choice is the *Nesterov's Accelerated Gradient*, both in their stochastic sub-gradient variant.

3.2.1 Classical Momentum

Classical Momentum (CM), also known as *Heavy Ball Method* (HB), is a momentum technique originally introduced by Polyak. This method accumulates a *velocity vector* in directions of persistent reduction in the objective across iterates [5]. Equivalently, we can see CM/HB in a different way, as using the previous two iterates when computing the next one [6].
The weight update works as follows:

$$\begin{cases} \mathbf{v}_{k+1} = \alpha \mathbf{v}_k - \eta \mathcal{G}(\mathbf{W}_k, \xi_k) \\ \mathbf{W}_{k+1} = \mathbf{W}_k + \mathbf{v}_{k+1} \end{cases} \tag{10}$$

where $\mathbf{v}$ is the *velocity vector* and $\alpha \in [0, 1] \subseteq \mathbb{R}$ is the *momentum coefficient* (an hyper-parameter). While η and $\mathcal{G}(\mathbf{W}_k, \xi_k)$ are the usual stepsize and stochastic sub-gradient.

Equivalently, we can write this update in one line, not showing the velocity vector explicitly, but making more visible the fact that the previous two iterates are taken into account:

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \eta\mathcal{G}(\mathbf{W}_k, \xi_k) + \alpha(\mathbf{W}_k - \mathbf{W}_{k-1}) \tag{11}$$

3.2.2 Nesterov's Accelerated Gradient

Nesterov's Accelerated Gradient (NAG) differs from CM/HB (Section 3.2.1) from the fact that the (sub)-gradient is not computed from the current position $\mathbf{W}_k$, but from a "look-forward" position $\mathbf{W}_k + \alpha\mathbf{v}_k$.
The weight update then becomes:

$$\begin{cases} \mathbf{v}_{k+1} = \alpha\mathbf{v}_k - \eta\mathcal{G}(\mathbf{W}_k + \alpha\mathbf{v}_k,\ \xi_k) \\ \mathbf{W}_{k+1} = \mathbf{W}_k + \mathbf{v}_{k+1} \end{cases} \tag{12}$$

As we can see, the update itself is the same as (10), it's the velocity vector that is computed in a different way.
Again, equivalently, we can write (12) without showing explicitly the velocity vector, with the help of an auxiliary variable $\mathbf{V}$:

$$\begin{cases} \mathbf{V}_{k+1} = \mathbf{W}_k - \eta\mathcal{G}(\mathbf{W}_k, \xi_k) \\ \mathbf{W}_{k+1} = \mathbf{V}_{k+1} + \alpha(\mathbf{V}_{k+1} - \mathbf{V}_k) \end{cases} \tag{13}$$

3.2.3 Unified view of momentum methods

We present now the *Unified Momentum* (UM) [6]. Since the choice of (sub)-gradient or stochastic (sub)-gradient is irrelevant to the discussion of the method, we denote by $\mathcal{G}(\mathbf{W})$ either one or the other. As always, $\eta \in [0, 1]$ is the learning rate and $\alpha \in [0, 1)$ is the momentum coefficient. Furthermore, let $s \geq 0$ be a new hyper-parameter that will make UM equivalent to CM/HB, NAG or GD/SGD.
The weight update of UM works as follows:

$$\begin{cases} \mathbf{Y}_{k+1} = \mathbf{W}_k - \eta \cdot \mathcal{G}(\mathbf{W}_k) \\ \mathbf{Y}^s_{k+1} = \mathbf{W}_k - s \cdot \eta \cdot \mathcal{G}(\mathbf{W}_k) \\ \mathbf{W}_{k+1} = \mathbf{Y}_{k+1} + \alpha(\mathbf{Y}^s_{k+1} - \mathbf{Y}^s_k) \end{cases} \tag{14}$$

with $\mathbf{Y}^s_0 = \mathbf{W}_0$. The momentum term uses the auxiliary sequence $\{\mathbf{Y}^s_k\}$, whose updates have a possibly different stepsize than those of $\{\mathbf{Y}_k\}$, depending on the value of s. In [6] it shown how setting $s = \frac{1}{1-\alpha}$ is equivalent to having a GD/SGD with learning rate equal to $\frac{\eta}{1-\alpha}$. It is simple math, but it is not in the interest of this report to show the steps to derive it. Moreover, it is also possible to simply set $\alpha = 0$ to delete the effect of $\{\mathbf{Y}^s_k\}$ and have a GD/SGD with the learning rate η of choice.
Instead, it is straight forward to see that if $s = 0$, UM is equivalent to CM/HB,

and if $s = 1$, it is equivalent to NAG.

3.2.4 Convergence analysis of stochastic CM/HB and NAG

To analize the convergence rate of these algorithms (Sections 3.2.1, 3.2.2), we will follow the analysis presented in [6], which proposes a unified analysis for the convergence of stochastic momentum methods as seen in the framework of *Unified Momentum* (we will refer to it as *Stochastic UM* (SUM)).
Let $\mathcal{G}(\mathbf{W}, \xi)$ denote a *stochastic* sub-gradient.

Theorem 1 (Convergence rate of SUM) *Suppose $J(\mathbf{W})$ is a non-convex L-smooth function, $\mathbb{E}[\|\mathcal{G}(\mathbf{W}, \xi) - \nabla J(\mathbf{W})\|^2] \leq \delta^2$ and $\|\nabla J(\mathbf{W})\| \leq G$. For any constant $C > 0$, let update (14) for t iterations.*
By setting $\eta = \min\{\frac{1-\alpha}{2L}, \frac{C}{\sqrt{t+1}}\}$ we have

$$\min_{k=0,\dots,t} \mathbb{E}[\|\nabla \mathcal{L}(\mathbf{W}_k)\|^2] \leq \frac{2(\mathcal{L}(\mathbf{W}_0) - \mathcal{L}_*)(1-\alpha)}{t+1} \max\{\frac{2L}{1-\alpha}, \frac{\sqrt{t+1}}{C}\} + \frac{C}{\sqrt{t+1}} \frac{L\alpha^2((1-\alpha)s-1)^2(G^2+\delta^2) + L\delta^2(1-\alpha)^2}{(1-\alpha)^3} \tag{15}$$

Note: the main difference in the convergence bound lies in the term $L\alpha^2((1-\alpha)s-1)^2(G^2+\delta^2)$, which is equal to $L\alpha^2$ for stochastic CM/HB, and to $L\alpha^4$ for stochastic NAG.

Unfortunately, our objective function is non-differentiable due to the L1 norm, therefore, we cannot state that our algorithms will converge with the rate shown in Theorem 1. Anyway, by giving up the regularization, we would get an objective function coinciding with the MSE (whose properties were described in Section 2.2.2), that respects indeed the premises of Theorem 1 and we could prove the convergence of CM/HB and NAG.
In practice, it is very common to use CM/HB and NAG in the training of neural networks with non-convex and non-differentiable (as well as differentiable) objective functions, without theoretical guarantees of convergence. This is instead shown empirically, by properly setting the learning rate, the weight initialization and considering that, as stated in Section 2.2.3, it is very unlikely to get some weights to be exactly 0, therefore our objective function is going to be differentiable for the most time. In case of an exact 0, we are going to use the sub-gradient and manually set $\frac{\partial \|w_{ij}\|_1}{\partial w_{ij}} = 0$.

4 M2: L2 linear regression (least squares)

4.1 Least squares

Least Squares can be explained in simple terms as the problem of determining the best fitting line to the data. Given that, for $n \in \{1, \dots, N\}$, the vectors $\mathbf{a}_n$ and $\mathbf{b}$ (*target vector*) are observed, the goal is to find the coefficients x_n such that $\mathbf{a}_1 x_1 + \cdots + \mathbf{a}_N x_N = \mathbf{b}$.
This problem can be expressed in the following form:

$$\min_{\mathbf{x} \in \mathbb{R}^n} \|A\mathbf{x} - \mathbf{b}\| \tag{16}$$

where $A = [\mathbf{a}_1 \; \mathbf{a}_2 \; \cdots \; \mathbf{a}_N]$ and $\mathbf{x} = [x_1 \; x_2 \; \cdots \; x_N]^T$.

4.2 SVD

The *Singular Value Decomposition* (SVD) allows us to factorize all matrices (square or rectangular) as a product of other matrices. In particular, each matrix A can be expressed as $A = U\Sigma V^T$ where the columns of U (left singular matrix) and V (right singular matrix) form two orthonormal sets and the matrix Σ is diagonal with positive real entries (called *singular values*). A simple consequence of the orthogonality is that for a square and invertible matrix A, its corresponding inverse is $A^{-1} = V\Sigma^{-1}U^T$ [7].
Let's treat the rows of our matrix $A^{n \times d}$ as n points in a *d-dimensional* space, the problem is to find the best *k-dimensional* subspace w.r.t. the set of points. It means minimizing the sum of the square of the perpendicular distances of the points ($\mathbf{a}_i$) to the subspace. This problem is also called *best least squares fit.*
Considering projecting a point $\mathbf{a}_i$ onto a line through the origin, then

$$(\text{distance of } \mathbf{a}_i \text{ to the line})^2 = a_{i1}^2 + a_{i2}^2 + \cdots + a_{id}^2 - (\text{length of the projection})^2 .$$

To minimize the sum above, one could minimize $\sum_{i=1}^{n} \left(a_{i1}^2 + a_{i2}^2 + \cdots + a_{id}^2\right)$ minus the sum of the squares of the lengths of the projections of the points to the line. However, that summation is a constant which is independent of the line, so minimizing the sum of the squares of the distances is equivalent to maximizing the sum of the squares of the lengths of the projections onto the line. Similarly for best-fit subspaces, we could maximize the sum of the squared lengths of the projections onto the subspace instead of minimizing the sum of squared distances to the subspace [7].

4.2.1 Singular vectors

We now define the *singular vectors* of an $n \times d$ matrix A.
Consider the best fit line through the origin and let $\mathbf{v}$ be a unit vector along this line. The length of the projection of $\mathbf{a}_i$ onto $\mathbf{v}$ is $\|\mathbf{a}_i \cdot \mathbf{v}\|$. From this we see that the squared sum of the length of the projections is $\|A\mathbf{v}\|^2$ and the best fit line is the one maximizing this quantity and hence minimizing the sum of the

squared distances of the points to the line. With this in mind we define the first *(right) singular vector* of A as

$$\mathbf{v}_1 = \underset{\|\mathbf{v}\|=1}{\text{argmax}} \|A\mathbf{v}\| \ . \tag{17}$$

The value $\sigma_1(A) = \|A\mathbf{v}_1\|$ is the first *singular value*[1] of A. Iteratively, we use a greedy approach to find the best fit k-dimensional subspace for a matrix A, by using the previous $\mathbf{v}_i$. The process stops when we have found $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_r$ as singular vectors and

$$\underset{\mathbf{v}\perp\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_r;\ \|\mathbf{v}\|=1}{\arg\max} \|A\mathbf{v}\| = 0 \ .$$

Theorem 2 *Let A be an $n \times d$ matrix where $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_r$ are the singular vectors defined above. For $1 \le k \le r$, let V_k be the subspace spanned by $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$. Then for each k, V_k is the best-fit k-dimensional subspace for A.*

Theorem 3 . *Let A be a rank r matrix. The left singular vectors of A, $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_r$ are orthogonal.*

4.2.2 Decomposition

Let A be an $n \times d$ matrix with right singular vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_r$ and corresponding singular values $\sigma_1, \sigma_2, \ldots, \sigma_r$ (found as explained in Sec. 4.2.1). Then, by taking

$$\mathbf{u}_i = \frac{1}{\sigma_i} A\mathbf{v}_i, \ \forall i \in [1, r] \subseteq \mathbb{N} \tag{18}$$

as the left singular vectors, the matrix A can be decomposed into a sum of rank one matrices as

$$\begin{aligned} A &= \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \\ &= U\Sigma V^T \end{aligned} \tag{19}$$

4.3 Solving least squares with SVD decomposition

Consider the least squares (LS) problem of eq. 16 (reported here as well)

$$\min_{\mathbf{x}\in\mathbb{R}^n} \|A\mathbf{x} - \mathbf{b}\|$$

where $A \in \mathbb{R}^{m\times n}$ and $A = U\Sigma V^T$ is its SVD decomposition. From this we can state:

$$\|A\mathbf{x} - \mathbf{b}\| = \|U\Sigma V^T\mathbf{x} - \mathbf{b}\| \ .$$

[1]In the SVD decomposition of A ($A = U\Sigma V^T$), σ_1 is the value on the top-left corner of the diagonal matrix Σ.

Since multiplying by an orthogonal matrix preserves the norm, and from the definition of orthogonality we know that, if a matrix B is orthogonal, $B^T = B^{-1}$ and $B^TB = BB^T = I$, we proceed as follows:

$$\|U^T(U\Sigma V^T\mathbf{x} - \mathbf{b})\| = \|U^TU\Sigma V^T\mathbf{x} - U^T\mathbf{b}\| = \|\Sigma V^T\mathbf{x} - U^T\mathbf{b}\| = \|\Sigma\mathbf{y} - U^T\mathbf{b}\|$$

setting $\mathbf{y} = V^T\mathbf{x}$.
Note that

$$\Sigma\mathbf{y} = [\sigma_1 y_1 \ \sigma_2 y_2 \ \cdots \ \sigma_n y_n \ 0 \ \cdots \ 0]^T$$

and

$$U^T\mathbf{b} = [\mathbf{u}_1^T\mathbf{b} \ \mathbf{u}_2^T\mathbf{b} \ \cdots \ \mathbf{u}_n^T\mathbf{b} \ \mathbf{u}_{n+1}^T\mathbf{b} \ \cdots \ \mathbf{u}_m^T\mathbf{b}]^T \ .$$

Therefore we have that

$$\|\Sigma\mathbf{y} - U^T\mathbf{b}\| = \sum_{i=1}^{r}\left(\sigma_i y_i - \mathbf{u}_i^T\mathbf{b}\right)^2 + \sum_{i=r+1}^{m}\left(\mathbf{u}_i^T\mathbf{b}\right)^2 \tag{20}$$

and its minimum, if all the σ's are different from 0 (and therefore A^TA is invertible since A has full column rank), is obtained setting

$$\begin{aligned} y_i &= \frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}, \ i = 1, \ldots, r \\ y_i &= \text{arbitrary}, \ i = r+1, \ldots, m \ . \end{aligned} \tag{21}$$

Finally, since $\mathbf{y} = V^T\mathbf{x}$ and V is orthogonal, we can find $\mathbf{x}$ as $\mathbf{x} = V\mathbf{y}$.
The minimum norm solution then becomes

$$\mathbf{x} = \sum_{i=1}^{r}\frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}\mathbf{v}_i \ . \tag{22}$$

Moreover, if A has full column rank, the solution is unique.

4.3.1 Properties and expected results

4.3.1.1 Tikhonov regularization As we said before, when all the singular values are different from 0, the solution is unique and it corresponds to 22. Let's now suppose that the $rank(A)$ (number of nonzero singular values) is not full, i.e. some $\sigma_i = 0$, that implies redundant models. In this case the minimum of the quadratic function is only positive semi-definite and therefore the solution is not unique. In many applications the most meaningful features correspond to the large singular values. A possible solution is the one that discourages solutions with large norm. The problem can be rewritten as:

$$\min_{\mathbf{x}\in\mathbb{R}^n}\left\|\begin{bmatrix} A \\ \alpha I \end{bmatrix}\mathbf{x} - \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}\right\|^2 \tag{23}$$

In particular, we can also rewrite it in a closed-form as

$$\begin{aligned}\begin{bmatrix} A \\ \alpha I \end{bmatrix}^{+} \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} &= \left(\begin{bmatrix} A \\ \alpha I \end{bmatrix}^{T} \begin{bmatrix} A \\ \alpha I \end{bmatrix} \right)^{-1} \begin{bmatrix} A \\ \alpha I \end{bmatrix}^{T} \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \\ &= \left(\begin{bmatrix} A^T & \alpha I \end{bmatrix} \begin{bmatrix} A \\ \alpha I \end{bmatrix} \right)^{-1} \begin{bmatrix} A^T & \alpha I \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \\ &= \left(A^T A + \alpha^2 I \right)^{-1} A^T \mathbf{b} \end{aligned} \tag{24}$$

Specifically we have:

$$\mathbf{z}^T \left(A^T A + \alpha^2 I \right) \mathbf{z} \geq \alpha^2 \mathbf{z}^T \mathbf{z} > 0 \text{ for all } \mathbf{z} \neq 0 \Longrightarrow \begin{bmatrix} A \\ \alpha I \end{bmatrix} \tag{25}$$

that has full column rank whenever the choice of $\alpha > 0$. This value depends on the amount of expected noise we have in the data. One of the best common practical solution in choosing the best value for α is performing a grid search.

4.3.1.2 Sensitivity

A matrix may have singular values that are not exactly equal to zero, but close to it. This is a problem because, in the setting of least squares, these values provide a higher contribution. One possible solution is to ignore all this values, simply rewriting the problem as:

$$\mathbf{x}_{reg} = \sum_{i=1}^{k} \frac{\mathbf{u}_i^T \mathbf{b}}{\sigma_i} \mathbf{v}_i, \quad (\text{ for a certain } k) \tag{26}$$

Eq. 26 however is not a real solution, but an approximation of it, that in many cases provides very good results anyway. The same issue appears when we apply a small perturbation on the input: since all the singular values change by the same amount, the relative error for the smallest ones is actually larger. In particular we know that the SVD algorithm is backward stable, this means that the computed SVD for a certain matrix A is comparable to the solution with a perturbation $(A + E)$ applied on it. The approximate error bounds for the computed singular values is stated in the following theorem:

Theorem 4 *Let σ_i be the singular values of A, and $\tilde{\sigma}_i$ those of $A + E$. Then, $\|\sigma_i - \tilde{\sigma}_i\| \leq \|E\|$*

where E is the perturbation applied to the input. Thus the largest singular values, those nearest σ_1, are computed with a high relative accuracy while the smallest ones may not be.

4.3.1.3 Computational Cost

The cost depends on the matrix partitioning. It can be partitioned row-wise (if the matrix is typically "tall and skinny") or column-wise (if the matrix is "short and fat").
In general, the computational cost of the SVD decomposition is equal to

$$O(mn \cdot min(n, m)) \tag{27}$$

5 Experiments

5.1 M1

In this sections we describe the experiments and the results obtained with M1: neural network with L1 regularization tested with SGD, heavy ball method and Nesterov's accelerated gradient.

5.1.1 Dataset

The dataset used for these experiments is the one used for the CUP of the *Machine Learning* course at University of Pisa for the academic year 2020-21. This dataset contains 10 numerical attributes and 2 numerical targets (for regression). Before using it in our experiments, it has been normalized column by column.

5.1.2 Experiments setup and results

M1 must be a neural network with L1 regularization. Apart from that, many different instances can be tested, with different architectures and hyperparameters. We exploited this through a grid search whose details are explained in what follows.

5.1.2.1 Grid search First of all, we have to divide the attributes into two categories:

- Model-related attributes: network architecture, activation function and L1 regularization coefficient (λ).
- Algorithm-related attributes: algorithm (SGD / HB / NAG), batch size, learning rate (η) and momentum coefficient (α).

The former directly influence the objective function itself (J), therefore, different configurations of these parameters realize different objective functions.
The latter, instead, have an influence on the algorithm used to minimize J. For each configuration of the model-related attributes, we performed a grid search on the algorithmic ones, and the values of those attributes are reported in Table 1.
To actually evaluate the convergence near the optimum point, we took the norm of the gradient as metric. More specifically, the trainings stopped when this norm was smaller than a certain threshold or when a defined maximum number of epochs was reached, acting as an emergency mechanism to kill the trainings that were taking to long. We also performed some experiments setting this threshold to zero, meaning that the algorithms were stopping only when reaching the specified number of epochs, effectively removing this "early stopping" mechanism.

Attribute	**Values**
Algorithm type	SGD, HB, NAG
Batch size	1, 100, full batch
Learning rate (η)	0.7, 0.5, 0.2, 0.05, 0.02, 0.01, 0.005, 0.001
Momentum coefficient (α)	0.0, 0.2, 0.5, 0.7, 0.9

Table 1: Values of the algorithmic attributes whose combinations have been tested in the grid search.

5.1.2.2 Performance metric To better analyze the convergence of the various algorithms, we take the *error gap* as main metric. This is defined as

$$\texttt{err\_gap} = \frac{J - J^*}{J^*} \tag{28}$$

where J is our objective function (Sec. 2.2) (associated to a specific configuration of the model-related attributes), which consists of MSE plus L1 regularization, while J^* is the optimal value of the same function. To select such optimal value (J^*), we then decided to let our algorithms run for "many"[2] epochs, and chose the best value we obtained.

5.1.2.3 Experiment 1 For the configuration of the model-related attributes listed in Table 2, we obtained the optimal value J^* with the combination of algorithmic attributes reported in Table 3, and such value is **0.0029**.

Table 2: Model-related attributes

Architecture	(10,5,2)
Activation function	sigmoid
L1 reg. coeff. (λ)	10^{-5}

Table 3: Alghoritmic attributes of the model that performed best

Algorithm type	SGD
Batch size	full batch
Learning rate η	0.02
Momentum (α)	0
Max Epochs	$480 \cdot 10^3$

First, we compare the three algorithms by taking, for each of them, the model that performed best (with the model-related attributes shown in Tab. 2) and plot their learning curves. For each loss value J_i in the curves, we take the gap $(J_i - J^*)/J^*$ and plot in logarithmic scale. The results are shown in Figure 3 and the characteristics of the models are stated in Table 4. It is possible to notice from the latter that all the three models share the same learning rate, showing that such particular value proved to work well, providing a good decrease in

[2]The number of epochs used to determine the optimum veries depending on the maximum number of epochs set for each experiment. Usually it is at least the double of the latter. For example, if in a certain experiment we set the maximum number of epochs to be 10000, then the optimum would be determined by letting the algorithms iterate for at least 20000 epochs, and then picking the lowest error we obtained.

the loss in the early stages of training (Fig. 3), leading to a lower error in the end. Higher values, instead, were usually causing instability. On the contrary, lower learning rates provide a stable curve but they might lead to a higher final error. To validate our performances, we compared our results with the ones obtained using PyTorch [8]. In particular, we ran each of those three models (Tab. 4), with the same model-related and algorithmic attributes, twice: once using our implementation and once using PyTorch [8]. All these learning curves are plotted on Figure 3.
As it is possible to see, our models' performances are comparable with those of the models implemented in PyTorch [8]. Furthermore, in both implementations, momentum methods reach a lower error than standard Gradient Descent, and, especially in the initial phase of training, they provide a faster convergence, as shown by the steeper learning curves. In this case, HB and NAG provide very similar results.

	$\boldsymbol{\eta}$	$\boldsymbol{\alpha}$	**Epochs**	**Final error**	**Final error gap**
Best SGD	0.2	0.0	17500	$3.1378 \cdot 10^{-3}$	$8.2014 \cdot 10^{-2}$
Best HB	0.2	0.7	17500	$2.9105 \cdot 10^{-3}$	$3.6146 \cdot 10^{-2}$
Best NAG	0.2	0.7	17500	$2.9137 \cdot 10^{-3}$	$4.7245 \cdot 10^{-2}$

Table 4: The characteristics of the models that performed best for each type of algorithm.

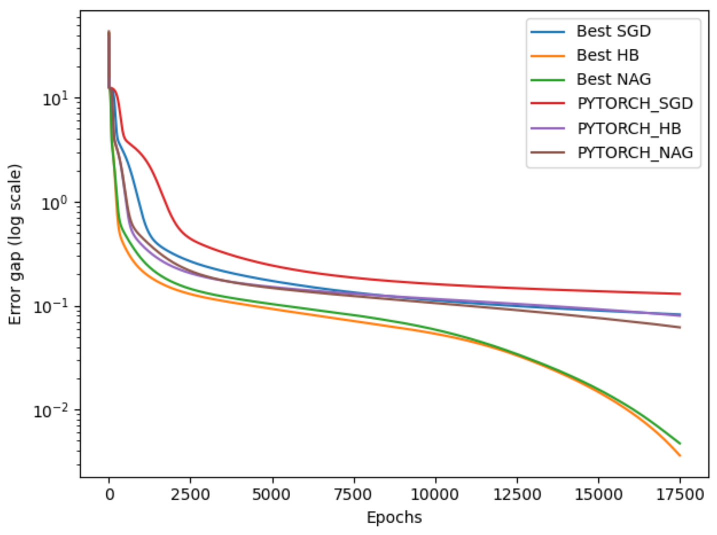

Figure 3: Learning curves of the models that, for each type of algorithm (SGD / HB / NAG), performed best with the model-related attributes of Tab. 2. In addition, as a comparison, there are the same models run using PyTorch [8]. The plots report the error gap with respect to the best optimal error J^*, in logarithmic scale.

5.1.2.4 Experiment 2 We decided to perform other experiments by changing the model-related attributes, referring to a different $J*$. The model-related attributes for this experiment are reported in Table 5 and the model which performed best with this configuration had the algorithmic attributes shown in Table 6. It reached an optimal value $J*$ equal to 0.00285 in 150000 epochs.
Table 7, instead, reports the model that performed best (with this configuration of model-related attributes) for each type of algorithm.
Here, all the models share again the same learning rate, further proving its effectiveness. Clearly, in both this experiment and the previous one, the trainings have been performed with a full batch setup, improving the stability of the curves and allowing for a larger learning rate than in an online/stochastic setting.
In this experiment, Figure 4 shows that the NAG is the algorithm which performed best among the others. Furthermore, towards the end of the curves. it is also possible to notice that SGD starts to have a slower convergence with respect to the HB, as could be expected. Also, in the initial phases of training, HB shows indeed a faster convergence than SGD. One more noticeable thing is that the momentum coefficient (α) of HB is relatively small, but still this proved to be the best configuration of algorithmic parameters for this type of algorithm (in the setting described in Tab. 5).

Table 5: Model-related attributes

Architecture	(10,20,5,2)
Activation function	sigmoid
L1 reg. coeff. (λ)	10^{-5}

Table 6: Alghoritmic attributes

Algorithm type	SGD
Batch size	full batch
Learning rate η	0.2
Momentum (α)	0.7
Max Epochs	$150 \cdot 10^3$

	$\boldsymbol{\eta}$	$\boldsymbol{\alpha}$	**Epochs**	**Final error**	**Final error gap**
Best SGD	0.2	0.0	100000	$2.9092 \cdot 10^{-3}$	$2.0802 \cdot 10^{-2}$
Best HB	0.2	0.2	100000	$2.8965 \cdot 10^{-3}$	$1.6336 \cdot 10^{-2}$
Best NAG	0.2	0.5	100000	$2.8579 \cdot 10^{-3}$	$2.7840 \cdot 10^{-3}$

Table 7: The characteristics of the models that performed best for each type of algorithm.

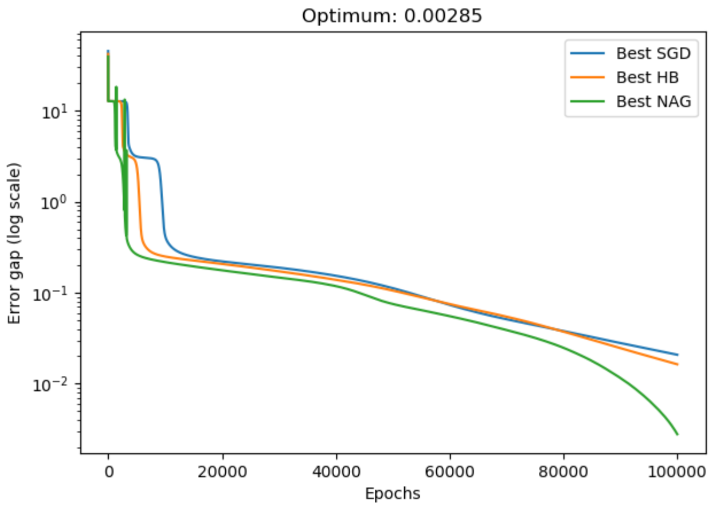

Figure 4: Learning curves of the models that, for each type of algorithm (SGD / HB / NAG), performed best with the attributes of Tab. 7. The plots report the error gap with respect to the best optimal error J^*, in logarithmic scale.

5.1.2.5 Experiment 3 This experiments is carried out using only one hidden layer (as Experiment 1 in Paragraph 5.1.2.3), but, instead of constricting the input of size 10 into a hidden layer of size 5, this time the hidden layer is much larger (50 units), creating a neural network with a total of 62 units, which is larger than the one of Experiment 2 (Paragraph 5.1.2.4), even though the

latter had one layer more.
Regarding the regularization coefficient, in this case it is set to 0.001, while in the previous two experiments it was 10^{-5}.
Looking at the learning curves of Figure 5, it is possible to observe how in this case the 3 algorithms have distinct performances, NAG being the best, followed by HB and lastly by SGD, which again it was in line with what we expected from the theory. In this case, though, the learning rate of the models is much smaller than in the previous experiments.

Table 8: Model-related attributes

Architecture	(10,50,2)
Activation function	sigmoid
L1 reg. coeff. (λ)	0.001

Table 9: Alghoritmic attributes

Algorithm type	NAG
Batch size	full batch
Learning rate η	0.01
Momentum (α)	0.9
Max Epochs	$30 \cdot 10^3$

	$\boldsymbol{\eta}$	$\boldsymbol{\alpha}$	**Epochs**	**Final error**	**Final error gap**
Best SGD	0.01	0.0	15000	$2.0339 \cdot 10^{-2}$	$3.5598 \cdot 10^{-1}$
Best HB	0.01	0.7	15000	$1.5929 \cdot 10^{-2}$	$6.1977 \cdot 10^{-2}$
Best NAG	0.01	0.7	15000	$1.5204 \cdot 10^{-2}$	$1.3608 \cdot 10^{-2}$

Table 10: The characteristics of the models that performed best for each type of algorithm.

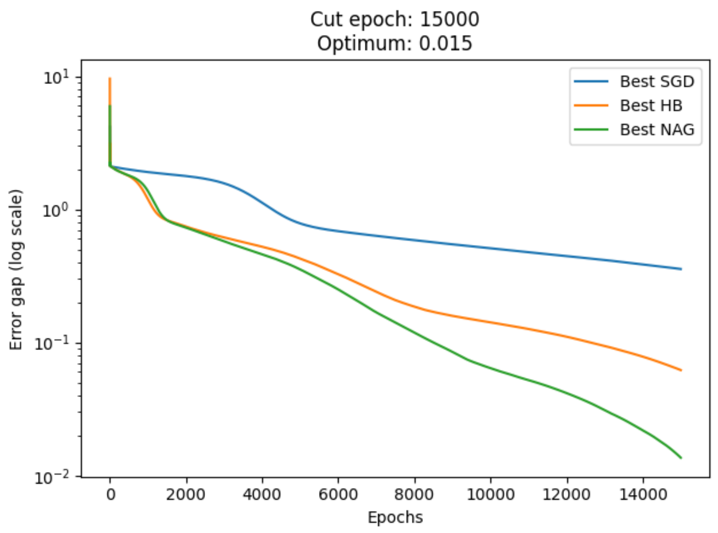

Figure 5: Learning curves of the models that, for each type of algorithm (SGD / HB / NAG), performed best with the attributes of Tab. 10. The plots report the error gap with respect to the best optimal error J^*, in logarithmic scale.

5.1.2.6 Experiment 4 Here, the model-related attributes configuration (Tab. 11) is similar to the one used in the previous experiment (Tab. 9), the only difference is the regularization coefficient, which passes from 10^{-3} to 10^{-5}. This led to having the model that gave the optimal error $J*$ (Tab. 12) to be substantially the same as in the previous experiment, with the only difference being the maximum number of epochs. Not only that, but also the models that performed best for each type of algorithm, described in Table 12, are the same as in Table 9, with again the only difference being the maximum number of epochs. The curves in Figure 6 show again distinct behaviours, even though it is worth pointing out that HB and NAG are very similar in the early phases of training. Overall, the models were robust to this change in the L1 coefficient (λ), because the best ones with $\lambda = 10^{-5}$ have the same configuration as the ones that performed best with $\lambda = 10^{-3}$ (being the rest of the setting unchanged).

Table 11: Model-related attributes

Architecture	(10,50,2)
Activation function	sigmoid
L1 reg. coeff. (λ)	1^{-5}

Table 12: Alghoritmic attributes

Algorithm type	NAG
Batch size	full batch
Learning rate η	0.01
Momentum (α)	0.9
Max Epochs	$50 \cdot 10^3$

	η	α	**Epochs**	**Final error**	**Final error gap**
Best SGD	0.01	0.0	20000	$4.9589 \cdot 10^{-3}$	$1.2703 \cdot 10^{-1}$
Best HB	0.01	0.7	20000	$4.6462 \cdot 10^{-3}$	$5.5956 \cdot 10^{-2}$
Best NAG	0.01	0.7	20000	$4.4639 \cdot 10^{-3}$	$1.4537 \cdot 10^{-2}$

Table 13: The characteristics of the models that performed best for each type of algorithm.

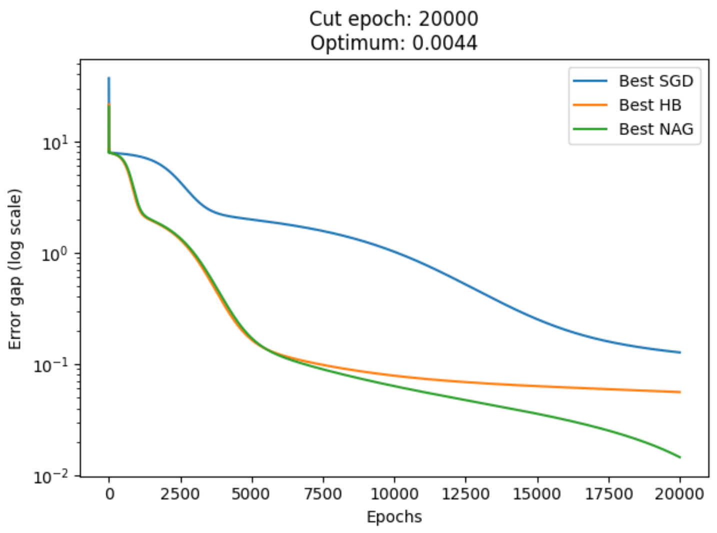

Figure 6: Learning curves of the models that, for each type of algorithm (SGD / HB / NAG), performed best with the attributes of Tab. 13. The plots report the error gap with respect to the best optimal error J^*, in logarithmic scale.

5.1.2.7 Experiment 5 In this experiment we further increased the number of layers, but keeping them relatively compact.
In this case the momentum-based algorithms are again better than Gradient Descent, providing a lower error, but especially a faster convergence. Anyway, this time HB and NAG show an almost identical training curve.

Table 14: Model-related attributes

Architecture	(10,10,10,5,2)
Activation function	sigmoid
L1 reg. coeff. (λ)	0.001

Table 15: Alghoritmic attributes

Algorithm type	NAG
Batch size	full batch
Learning rate η	0.01
Momentum (α)	0.5
Max Epochs	$25 \cdot 10^3$

	η	α	**Epochs**	**Final error**	**Final error gap**
Best SGD	0.01	0.0	20000	$3.9694 \cdot 10^{-2}$	$4.9194 \cdot 10^{-3}$
Best HB	0.01	0.5	20000	$3.9645 \cdot 10^{-2}$	$3.6799 \cdot 10^{-3}$
Best NAG	0.01	0.5	20000	$3.9648 \cdot 10^{-2}$	$3.7661 \cdot 10^{-3}$

Table 16: The characteristics of the models that performed best for each type of algorithm.

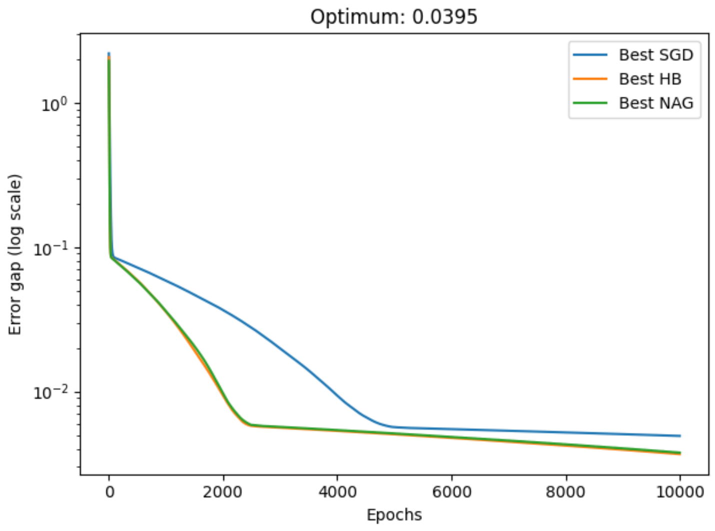

Figure 7: Learning curves of the models that, for each type of algorithm (SGD / HB / NAG), performed best with the attributes of Tab. 16. The plots report the error gap with respect to the best optimal error J^*, in logarithmic scale.

5.1.2.8 Variability analysis During our experiments, we noticed that it was a bit difficult to get very low error gaps (e.g. 10^{-4} or smaller), therefore, we started investigating the situation a little bit.
First of all, let us recall that the error gap is computed as in Eq. 28, which is reported also here for practicality:

$$\texttt{err\_gap} = \frac{J - J^*}{J^*}$$

where $J*$ is the optimal error and J is a certain value of our objective function. For definition, the optimum is the lowest value of the objective function that we can get, thus we can state that $J > J^*$ and we can represent J as $J = J^* + \epsilon$, where $\epsilon > 0$. Now, taking as reference our Experiment 1 (Paragraph 5.1.2.3), we set $J^* = 0.0029$. If we want a desired gap of 10^{-4}, for example, we can find the value of ϵ by substituting these numbers into the definition of the gap (Eq. 28) and solving the equation:

$$err\_gap = \frac{J^* + \epsilon - J^*}{J^*} \;;$$
$$\epsilon = err\_gap \cdot J^* = 10^{-4} \cdot 0.0029 = 2.9 \cdot 10^{-7}$$

This means that, in order to get that desired error gap, having a $J^* = 0.0029$, our objective function can differ from J^* by a maximum of $2.9 \cdot 10^{-7}$. It is clear that the margin is very narrow.
Obviously, when training the same model multiple times, with the same model-related and algorithmic hyper-parameters, the final outcome is not going to be identical every time, but there will be some minor differences in performance due to some stochasticity in the process (i.e. random initialization of the weights, minibatches, etc.). What we observed is that the average variability in the performances of the models makes various runs differ by an amount which is usually larger than what it would be needed to get an error gap that stays in the same order of magnitude. This means that if a particular training of a model reaches an error gap in the order of 10^{-4}, there will be also many other trainings of the same model, with the same hyper-parameters, that will give an error gap in a higher order of magnitude (e.g. 10^{-3}, 10^{-2}).
To support this, we performed some experiments running the training of various models multiple times in the same way, and analyzing the variability of their performance, both using our implementation and PyTorch [8]. As reference, we took the models reported in Table 4 of the Experiment 1 (Paragraph 5.1.2.3).

Best SGD: First of all, in this case PyTorch provided a slightly higher variability, in fact, the results of Tab. 17 have a variance of $\mathbf{1.09842 \cdot 10^{-9}}$, while the ones of Tab. 18 have a variance of $\mathbf{4.37761 \cdot 10^{-9}}$. The impact of such values of variance on the error gaps, though, are a bit implicit. For this reason we performed the following analysis: for each table (17 and 18) we computed the error gap (Eq. 28) between each pair of values, getting a measure of the average gap between two distinct runs of the same model (with identical model-related and algorithmic hyper-parameters). In the case of our implementation, we obtained $\mathbf{1.4272 \cdot 10^{-2}}$, while for PyTorch we got $\mathbf{2.7594 \cdot 10^{-2}}$.

Table 17: Ours (SGD)

Final error
0.0031513085639242808
0.0031621364188944676
0.0031957050129243444
0.0032066348271736407
0.003244120892298901

Table 18: PyTorch's (SGD)

Final error
0.0031149478163570166
0.0031647056636959314
0.0031747056636959314
0.0032742032781243324
0.0032854508608579636

Best HB: In this case, the situation is opposite, with our implementation getting an average gap of $\mathbf{3.4025 \cdot 10^{-2}}$, while PyTorch got $\mathbf{3.0825 \cdot 10^{-2}}$ (with corresponding variances of $\mathbf{5.86529 \cdot 10^{-9}}$ and $\mathbf{5.02547 \cdot 10^{-9}}$ respectively).

Table 19: Ours (HB)

Final error
0.0029224178662203147
0.0029442515704027096
0.0029892755892271107
0.003091623224066157
0.0031112756940695676

Table 20: PyTorch's (HB)

Final error
0.0029356973245739937
0.0030396776273846626
0.0030737947672605515
0.0031159548088908195
0.0031370907090604305

Best NAG: For this algorithm, the average gap of our implementation was $\mathbf{2.43817 \cdot 10^{-2}}$ and for PyTorch it was $\mathbf{2.25413 \cdot 10^{-2}}$ (with respective variance values of $\mathbf{3.25085 \cdot 10^{-9}}$ and $\mathbf{2.18588 \cdot 10^{-9}}$).

Table 21: Ours (NAG)

Final error
0.0029346457327154418
0.002944200841692842
0.0029540417195065135
0.003013343197455308
0.003086510588386424

Table 22: PyTorch's (NAG)

Final error
0.003016267903149128
0.0030396776273846626
0.003096954431384802
0.0031226903665810823
0.00313621130771935

As it is possible to notice, the variance/average gap of our implementation is always comparable with the ones of PyTorch. These average gaps, though, are in all three cases in the order of 10^{-2}. This explains why it was difficult to get consistently gaps in the order of 10^{-4} or lower between different models[3].

5.1.2.9 Conclusion After all the previous experiments, it is possible to notice that the different algorithms (SGD, NAG, HB) respect what predicted by the theory. In particular, it is possible to state that NAG has definitely a faster convergence with respect to the other algorithms (SGD and HB), even though in Experiment 1 and 5 (Par. 5.1.2.3, 5.1.2.7) its performances were pretty comparable with the ones of HB. In turn, HB had a faster convergence than SGD in all the experiments.
Finally, we would affirm that in general, for this kind of optimization problems, the usage of momentum is definitely worth trying.

5.2 M2

In this sections we describe the experiments and the results obtained with M2 (least squares with SVD).

[3]With "different models" we mean that there was one which gave us the value of $J*$ and others that were compared with it. The latter had different hyper-parameters (either model-related, algorithmic or both).

5.2.1 Dataset

The dataset we used for experiments contains data regarding the weather in Szeged (a city in Hungary)[9]. Some very simple preprocessing has been performed, such as dropping some columns (mainly textual ones) or changing the data type of some attributes (e.g. date format into numerical/integers etc). Finally the dataset has been normalized column by column. At this point it contains **96453 rows** and the following columns: **temperature** (float), **apparent temperature** (float), **humidity** (float), **wind speed** (float), **atmospheric pressure** (float), **month** (integer), **day** (integer), **hour of the day** (integer), **rain** (binary: 0 or 1) and **snow** (binary: 0 or 1).
The column containing the temperatures is used as target, in the form of a vector that we will call b, while the rest of the dataset, represented as a matrix called A, is used to regress the temperatures. This matrix has 9 columns and full rank [4].

5.2.2 Least Square - SVD

First, let's recall the formulation of the least squares problem:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \|A\mathbf{x}-\mathbf{b}\| \tag{29}$$

We define:

$$\mathbf{r} = A\tilde{\mathbf{x}} - \mathbf{b} \tag{30}$$

as the *residual*, that in some way measures how well our solution $\tilde{\mathbf{x}}$, multiplied by A, approximate the target vector $\mathbf{b}$.
By executing our implementation using the dataset of Sec. 5.2.1, the relative error between $A\tilde{\mathbf{x}}$ and $\mathbf{b}$ is defined as

$$E = \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} = \frac{\|A\tilde{\mathbf{x}}-\mathbf{b}\|}{\|\mathbf{b}\|} \tag{31}$$

and is equal to 0.0627.
Furthermore, the maximum difference between any component in the approximated solution $A\tilde{\mathbf{x}}$ and the corresponding element in the actual target vector $\mathbf{b}$ is at most 0.00103.
Considering that A and $\mathbf{b}$ are normalized, i.e. all of their components are within the interval [0, 1], it is possible to observe that $\tilde{\mathbf{x}}$ produces a residual which is relatively small.
Nevertheless, this does not tell how close $\tilde{\mathbf{x}}$ is to the actual solution (namely $\mathbf{x}$) of the problem formulated as in Eq. (29). Recalling that we're analyzing an optimization problem, it might be worth to look at the gradient. If we define $f(\mathbf{x}) = \frac{1}{2}\|A\mathbf{x}-\mathbf{b}\|^2$, then $\nabla f(\tilde{\mathbf{x}}) = A^T A\tilde{\mathbf{x}} - A^T\mathbf{b}$. To check how close is the computed solution $\tilde{\mathbf{x}}$ to the actual one $\mathbf{x}$, it is possible to use the following

[4]we observed this property with the SVD decomposition provided in scipy [10].

relation involving the gradient:

$$\frac{\|\mathbf{x}-\widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \kappa\left(A^T A\right) \frac{\left\|A^T A \widetilde{\mathbf{x}}-A^T \mathbf{b}\right\|}{\left\|A^T \mathbf{b}\right\|}=B \tag{32}$$

Where κ indicates the condition number and noticing that $\kappa(A^T A) = \kappa(A)^2$, where $\kappa(A) = \frac{\sigma_1}{\sigma_n}$ (σ_1 and σ_n are respectively the largest and smallest singular values of A).
The above bound B (defined in Eq. (32)), in our case, is equal to $7.3472 \cdot 10^{-12}$, and therefore it is possible to state that $\tilde{\mathbf{x}}$ is a good approximation of the true solution $\mathbf{x}$ of (29).

5.2.2.1 L2 regularization By recalling that we are analyzing a least squares problem defined as Eq. (29), in this section we investigate on the effects of adding the regularization term $\alpha^2\|\mathbf{x}\|^2$.
We define

$$\min_{\mathbf{x}_{L2} \in \mathbb{R}^n} \left\| \begin{bmatrix} A \\ \alpha I \end{bmatrix} \mathbf{x}_{L2} - \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \right\|^2 \tag{33}$$

as the original problem with the addition of the Tikhonov regularization. As before, we are interested in measure how well the computed solution $\tilde{\mathbf{x}}_{L2}$, multiplied by $\begin{bmatrix} A \\ \alpha I \end{bmatrix}$, where α is the regularization coefficient (hyper-parameter), approximate the target vector $\begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}$. The best value for α is chosen using a holdout validation method. First the original matrix A and the target vector $\mathbf{b}$ were split in two parts each (A_{tr}, A_{ts} and b_{tr}, b_{ts}) by using `train_test_split` method by scikit-learn [11]. Moreover, before the splitting, the rows were shuffled and a seed has been used to get replicable results.
Then we proceeded to solve the least square problem with Tikhonov regularization (34) on A_{tr}, i.e.

$$\min_{\mathbf{x}_{L2-tr} \in \mathbb{R}^n} \left\| \begin{bmatrix} A_{tr} \\ \alpha I \end{bmatrix} \mathbf{x}_{L2-tr} - \begin{bmatrix} \mathbf{b}_{tr} \\ 0 \end{bmatrix} \right\|^2 . \tag{34}$$

The computed solution of the problem (34) is referred to as $\tilde{\mathbf{x}}_{L2-tr}$ (the subscript tr is used to underline the fact that the solution has been computed using A_{tr} and b_{tr}).
We tested the performance of $\tilde{\mathbf{x}}_{L2\text{-}tr}$ using the test set (A_{ts} and $\mathbf{b}_{ts}$), exploiting the following metrics:

- Relative approximation error that measures how well the computed solution ($\tilde{\mathbf{x}}_{tr}$) on the available training data (A_{tr}), when used with new unseen test data A_{ts}, approximates the test target vector $\mathbf{b}_{ts}$:

$$E_{L2-ts} = \frac{\|A_{ts}\tilde{\mathbf{x}}_{L2-tr} - \mathbf{b}_{ts}\|}{\|\mathbf{b}_{ts}\|}$$

- Bound on the relative difference between $\tilde{\mathbf{x}}_{L2-tr}$ and the true solution (namely $\mathbf{x}_{L2-ts}$) of the test data [5]:

$$\frac{\|\mathbf{x}_{L2-ts} - \tilde{\mathbf{x}}_{L2-tr}\|}{\|\mathbf{x}_{L2-ts}\|} \leq \kappa(A_{ts}^T A_{ts}) \frac{\|A_{ts}^T A_{ts} \tilde{\mathbf{x}}_{L2-tr} - A_{ts}^T \mathbf{b}_{ts}\|}{\|A_{ts}^T \mathbf{b}_{ts}\|} = B_{L2-ts}$$

This procedure has been repeated for different values of α, namely: 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.01, 10^{-3}, 10^{-4} and 10^{-5}, and for each of them, we saved the minimum error for each of the metrics above and the α value that led to that error.
Our findings were the following:

- Minimum relative approximation error (1st metric):
  $\min E_{L2-ts} = 0.0628$
  Corresponding α: 0.9

- Minimum bound on the relative difference between $\tilde{\mathbf{x}}_{L2-tr}$ and the true solution $\mathbf{x}_{L2-ts}$ of the test data (2nd metric):
  $\min B_{L2-ts} = 0.2924$
  Corresponding α: 0.8

B_{L2-ts} is not as good as B, but still acceptable. After all, this behavior is to be expected for the following reasons:

- $\tilde{\mathbf{x}}_{L2-tr}$ is a regularized solution (unlike $\tilde{\mathbf{x}}$), therefore it does not optimize the problem freely, but its norm is pushed to be as small as possible, creating a compromise.

- B_{L2-ts} compares $\tilde{\mathbf{x}}_{L2-tr}$ and $\mathbf{x}_{L2-ts}$, the former is computed on the **training data** (A_{tr} and $\mathbf{b}_{tr}$), while the latter represents what would be the true solution of the **ts data** (A_{ts} and $\mathbf{b}_{ts}$) on which $\tilde{\mathbf{x}}_{L2-tr}$ is evaluated.

It is also worth considering that E_{L2-ts} is still low (0.0631) and therefore $\tilde{\mathbf{x}}_{L2-tr}$ is such that $A_{ts}\tilde{\mathbf{x}}_{L2-ts}$ is close to $\mathbf{b}_{ts}$.

As a further test, we tried to execute the same experiments on a perturbed input: $A_{noise} = A + N$, where N is a random matrix obtained using `numpy.random.normal` [12]. A_{noise} has then been split into A_{trn} and A_{tsn} (the suffix "n" in the subscript stands for "noise").
Let's define:

$$E_{L2-tsn} = \frac{\|A_{tsn}\tilde{\mathbf{x}}_{L2-trn} - \mathbf{b}_{ts}\|}{\|\mathbf{b}_{ts}\|}$$

and

$$B_{L2-tsn} = \kappa(A_{tsn}^T A_{tsn}) \frac{\|A_{tsn}^T A_{tsn} \tilde{\mathbf{x}}_{L2-trn} - A_{tsn}^T \mathbf{b}_{ts}\|}{\|A_{tsn}^T \mathbf{b}_{ts}\|} \geq \frac{\|\mathbf{x}_{L2-tsn} - \tilde{\mathbf{x}}_{L2-trn}\|}{\|\mathbf{x}_{L2-tsn}\|}$$

[5] $\mathbf{x}_{L2-ts} = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \left\| \begin{bmatrix} A_{ts} \\ \alpha I \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{b}_{ts} \\ 0 \end{bmatrix} \right\|^2$

where $\tilde{\mathbf{x}}_{L2-trn}$ indicates the computed solution of the least squares problem with L2 regularization on the noisy training data (A_{trn}), while $\mathbf{x}_{L2-tsn}$ represents what would be the true solution of the least squares problem with L2 regularization on the noisy test data (A_{tsn}).
We obtained the following results:

- $\min E_{L2-tsn} = 0.4899$ (corresponding α: 0.9)
- $\min B_{L2-tsn} = 0.0522$ (corresponding α: 0.9)

These metrics shows that $\tilde{\mathbf{x}}_{L2-trn}$ is still a decent approximation, also it is to be noted that, depending on different instances of N (N being random normal) the results can change noticeably (while still remaining in the same order of magnitude).

References

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[2] David Rumelhart, Geoffry Hinton, and Ronald Williams. Learning representations by back-propagating errors. 323:533–536, 1986.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] Xiaoyu Li and Francesco Orabona. On the convergence of stochastic gradient descent with adaptive stepsizes. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pages 983–992. PMLR, 16–18 Apr 2019.

[5] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, pages 1139–1147. PMLR, 2013.

[6] Tianbao Yang, Qihang Lin, and Zhe Li. Unified convergence analysis of stochastic momentum methods for convex and non-convex optimization. 04 2016.

[7] John Hopcroft and Ravi Kannan. *Computer Science Theory for the Information Age*, chapter 6, pages 2–3. 2012. `https://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/`.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[9] Historical weather dataset of szeged. `https://www.kaggle.com/shahmirali/historical-weather-dataset-of-szeged`. Accessed: 2021.

[10] Scipy svd. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.svd.html`.

[11] train_test_split by scikit-learn. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`.

[12] numpy.random.normal. `https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html`.