Computational Mathematics for Learning and Data Analysis

Gemma Ragadini

AY: 2023-2024

Contents

1 Introduction

As outlined in the project guidelines (ML Project 6) the goal of this project was to develop:

- **M1**: a neural network with piecewise-linear activation function and any regularization allowed;
- **M2**: a standard L_2 linear regression (least squares);
- **A1**: a standard momentum descent (heavy ball) approach applied to (M1);
- **A2**: an algorithm of the class of deflected subgradient methods, applied to (M1);
- **A3**: a basic version of the direct linear least squares solver applied to (M2).

2 Model 1: Neural Network

A Neural Network (NN), also known as a Multi-Layer Perceptron (MLP), is a computational model designed to produce a desired output from a given input (such as for tasks like classification or prediction) after undergoing a learning phase. During this phase, its internal parameters are adjusted using a learning algorithm. In more detail, an NN consists of multiple fully-connected layers. Each layer contains a set of weights $\mathbf{w}$ determined by the number of neurons (or computing units) and includes a bias b. A layer receives an input $\mathbf{x}$, either a sample if it's the first layer or the output from the preceding layer, and produces an output by combining the input with the weights (forming what is known as the *net*) and applying an *activation function* to introduce non-linearity.

$$\mathbf{net} = \mathbf{w}^T\mathbf{x} + b \tag{1}$$

$$\mathbf{o} = f_\sigma(\mathbf{net}) \tag{2}$$

As a result, it becomes feasible to formulate the complete equation of a neural network based on its architecture. For instance, consider the equation representing a neural network with one hidden layer and one output layer. Here, f_o and f_h represent the activation functions associated with the output and hidden layers respectively and the name of the function h derives from the fact that it's called the *hypothesis function.*

$$h(\mathbf{x}) = f_o(\mathbf{w}_o^T(f_h(\mathbf{w_h^T}\mathbf{x} + b_h)) + b_o) \tag{3}$$

As requested by the project assignment, the $ReLU$ function (piecewise linear and non-differentiable at 0) has been used as the activation function for the hidden layer.

$$ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

Without the addition of an activation function, a neural network wouldn't be able to solve more than linear problems. However, with this addition, the validity of the Universal Approximation Theorem is achieved ([**4**], 11.6). Our model will learn by leveraging a dataset: a collection of examples with corresponding labels that describe the desired outcomes.

$$\{(\mathbf{x}_i, \mathbf{d}_i)\}_{i=1}^{n} \tag{4}$$

where n is the amount of samples, and the label $\mathbf{d}_i$ can assume values according to the specific task. The Loss function chosen for this project is the *Mean Squared Error* (MSE):

$$E = \frac{1}{n}\sum_{i}^{n}(\mathbf{d}_i - h(\mathbf{x}_i))^2 \tag{5}$$

Where d_i is the label of the sample i and $h(\mathbf{x}_i)$ the correspondent value produced by the model. The objective is to approach the minimum of the loss function by adjusting the parameters of the network, weights, and biases.
L2 regularization was involved, which consists in adding to the loss function the term $\frac{\lambda}{2}\|\mathbf{w}\|$, where λ is the regularization parameter, in order to simplify the model and enable better generalization, penalizing weights with a larger norm. So the error of the model will be:

$$E = \frac{1}{n}\sum_{i}^{n}(\mathbf{d}_i - h(\mathbf{x}_i))^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2 \tag{6}$$

2.1 Implementation Overview

I chose to use the `ML-CUP-23` dataset from Machine Learning course (regression task), which is why I opted to implement a neural network with 1 hidden layer and to fix the problem to optimize as follows: *topology* $= [10, 100, 3]$ and $\lambda = 10^{-5}$. I wrote the code in Python and created classes `m1` and `m2` for the models. Moreover a `gradient` class was created, composed of matrices with dimensions suitable for containing the weights of the network, as well as the gradient, and finally the deflection. I implemented the ability to learn through minibatches, to plot the objective function against the number of epochs and to create and plot a tomography. There is a text file `"config_NN.txt"`, its purpose is to set the parameters of the model.

2.2 Algorithm 1: Momentum Descent (Heavy Ball) Approach

The main idea of gradient descent approach is to use $\Delta\mathbf{w} = -\nabla L(\mathbf{w})$ into the descent rule

$$\mathbf{w}^{i+1} = \mathbf{w}^i + \eta\Delta\mathbf{w}^i$$

The rule used to update the weigths of the neural network using a momentum descent approach is:

$$\mathbf{w}^{i+1} = \mathbf{w}^i - \eta\nabla L^i + \beta\nabla L^{i-1} \tag{7}$$

Where $\mathbf{w}$ is the nn's weigths matrix, η is the stepsize, β is the momentum parameter and ∇L^i is the gradient of the Loss function: the matrix of the dervatives $\frac{\partial L^i}{\partial w^i_{jk}}$, the derivatives of the Loss function with respect to each component of the weigths matrix. The addition of momentum in the learning rule allows considering the descent direction of the previous iteration and leads to better convergence (less zig-zag in the descent), depending on the value of β. At each iteration, the new gradient of the Loss function with respect to the (j, k) weigth (the weigth from unit j to unit k) is calculated through error *backpropagation*, omitting i for the i^{th} iteration:

$$\frac{\partial L}{\partial w_{jk}} = \delta_k o_j \tag{8}$$

where δ is different if the event j is a unit of output (case 1) or it's hidden layer (case 2). Let's call f_j the activation function correspondent to the layer of unit j and $net_j = \sum_s w_{sj}o_s$, the weighted sum computed by j before the application of f_j (index t iterates on the units of the k's layer).

$$\begin{aligned} &(1) \quad \delta_j = (d_j - o_j)f'_j(net_j) \\ &(2) \quad \delta_j = \left(\sum_t \delta_t w_{jt}\right) f^i_j(net_j) \end{aligned} \tag{9}$$

Fundamentally, the gradient is computed layer by layer, starting from the output layer and will represent for each weigth, its contribution to the error. Summarizing, the proposed approach to the neural network is:

```
Algorithm 1: Momentum Descent Approach
Input: initial $w$, initial $\nabla L$, $\eta$, $\beta$
for every example input $x$ do
    compute o and L
    using backprop compute $\nabla L$
    using (7) update $w$
```

2.2.1 What to expect from the algorithm

The loss function that we aim to minimize is not differentiable, as the activation function used is not differentiable (ReLU is not differentiable at $x = 0$), so I choose to compute a subgradient. There are no guarantees that the chosen subgradient will provide a descent direction, and for this reason, we cannot expect great performance. However, using a small stepsize seems to work quite well, as can be seen from the tests. Additionally, the objective funtion is also not-convex, as a NN, but to provide evidence of its non-convexity, I plotted a tomography along a random direction, an array generated with $np.random.uniform(-1, 1)$ and normalized, where it can be seen that the function is not convex along that direction and, consequently, is not convex overall (shown in Figure 1). The tomography has been created with $\varphi_{w,d}(t) = L(x + \alpha d)$. As we can see

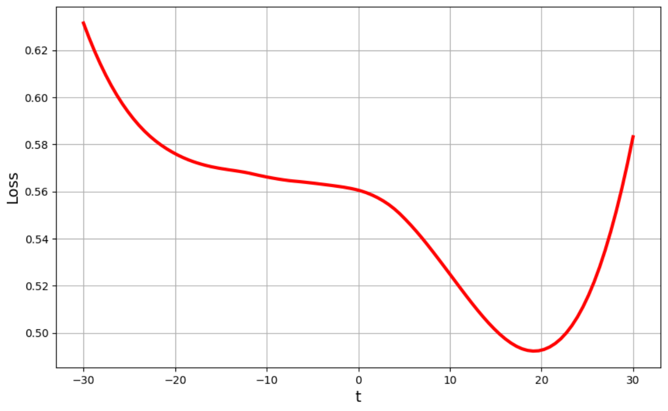

Figure 1: Tomography along a direction

from the tomography, the function is quasiconvex. To obtain this tomography, where a portion of non-convexity is evident, several attempts were necessary. This suggests that, while our objective function is not smooth, it might not be so different from a smooth function. We know that the Heavy-Ball (HB) algorithm for τ-convex and L-smooth objective function has *linear convergence* with optimal rate $r = \frac{\sqrt{k}-1}{\sqrt{k}+1}$, where $k = \frac{L}{\tau}$ ([**6**], Th. 3.15), choosing good parameters . Although our objective function is not differentiable, we expect it to behave similarly to a convex function since the ReLU introduces a point of non-differentiability, but we expect that regularization will mitigate this aspect. Due to L2 regularization, by the definition of τ-convexity, we expect our regularization parameter λ to influence r in a manner similar to how τ would. Due to these considerations, one would expect the convergence of the HB algorithm on the problem to be similar to linear convergence, as the problem is not very different from what one would encounter in the case of smoothness and convexity.

2.2.2 Implementation Overview

The following are the main implementation choices made for this algorithm. The algorithm is implemented into a class `a1` inheriting from class `m1`. `a1` consists of a main method `learning` wich calls the others: `backpropagation` and `update_weights`. `main1.py` calls the learning method of class `a1` if specified by command line. The parameters used are specified in a text file named `"config_a1.txt"` along with some model parameters. Those related to algorithm `a1` are: `stepsize` and `momentum`.

2.2.3 Considerations and Results

Using $stepsize = 10^{-3}$ and $momentum = 10^{-4}$ on dataset "ML-CUP23-TR.csv" heavy ball algorithm converges on model 1, with a trand shown in the graph (Figure 2), where the value of $\log(E_r)$ varies on 3000 epochs and E_r is the error relative to the minimum value found for the objective function (Loss added regularization) computed between algorithm a1 and a2, after running for 5000 epochs (0.177187).

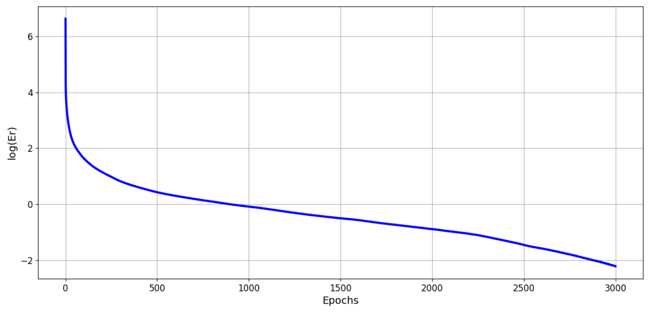

Figure 2: objective function varying with the number of epochs

Let's define *convergence rate* r of a sequence f^i to the optimal value f^*:

$$r = \lim_{i\to\infty} \frac{f^{i+1} - f^*}{(f^i - f^*)^p} \tag{10}$$

As can be seen from the Figure 2, the algorithm exhibits *sublinear convergence*, which, however, appears at the tail, to be quite similar to *linear convergence*. This seems consistent with the assumptions regarding the properties of the objective function. In Figure, the optimization curves using various algorithm parameter configurations are compared.

- *Config 1:* $stepsize = 0.0001$, $momentum = 0.00001$
- *Config 3:* $stepsize = 0.001$, $momentum = 0.0005$
- *Config 2:* $stepsize = 0.00015$, $momentum = 0.000015$

As we will see in 2.4, we are assuming that all algorithms converge to the same minimum. From the Figure 3, we can see that in each case, convergence is sublinear; there aren't significant performance differences, but the issue might be the speed of convergence.

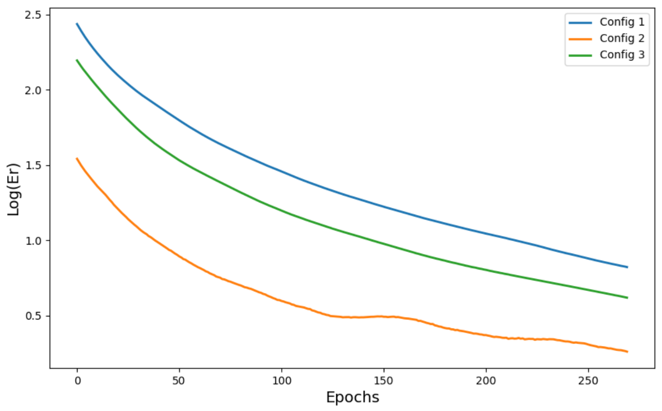

Figure 3: Comparison between different parameters configurations

2.3 Algorithm 2: A Deflected Subgradient Method

In this section, we address a simple algorithm from the class of subgradient methods, whose implementation is the result of studying and applying the methods discussed in [**1**], [**2**]. An algorithm from the class of deflected subgradient methods is used due to the non-differentiability of the function to minimize, as previously discussed. The arbitrary choice made regarding its value at the non-differentiable point is that it equals 1. Therefore, we will have:

$$\frac{\partial ReLu(x)}{\partial x} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases} \tag{11}$$

In the *backpropagation* process as in *Algorithm 1*, the subgradient is computed. However, an additional deflection is introduced, calculated at iteration i as follows:

$$\mathbf{d_i} = ||\alpha_i \mathbf{g_i} + (1-\alpha_i)\mathbf{g_{i-1}}||_2^2 \tag{12}$$

Where $\mathbf{g_i}$ is the subgradient, and α_i is

$$\alpha_i = \text{argmin}_{\alpha_j \in [0,1]} \, ||\alpha_i \mathbf{g_i} + (1-\alpha_i)\mathbf{g_i}||_2^2 \tag{13}$$

When, omitting i and defining $\mathbf{v} = \mathbf{g} - \mathbf{d}$, it is computed as follows:

$$||\alpha \mathbf{v} + \mathbf{d}||_2^2 = \alpha^2 ||\mathbf{v}||_2^2 + 2\alpha < \mathbf{v}, \mathbf{d} > + ||\mathbf{d}||_2^2$$

So, we can find the minimum as:

$$\alpha = -\frac{< \mathbf{v}, \mathbf{d} >}{||\mathbf{v}||_2^2} \ \textit{if } \mathbf{g} \neq \mathbf{d}$$

Otherwise, we can chose a value for α in $[0, 1]$; for the implementation I choose 0.5. If the minimum is not into the wished range, we can check values at the extremes. Moreover, *stepsize* has to be different for each iteration due to the non differentiability of the function, so I choose a stepsize rule similar to *Diminishing–Square Summable stepsize (DSS)* η, as follows:

$$\eta = \frac{\beta}{i} \tag{14}$$

By defining j as a counter starting from $j = 1$ for the number of consecutive non-improving iterations, and fixing k chosen a priori, j will be reset if its value exceeds k, and i is the counter to be incremented in that case. At the end the rule for updating weights at iteration i will be:

$$\mathbf{w_{i+1}} = \mathbf{w_i} - \eta_i \mathbf{d_i} \tag{15}$$

Algorithm 2: Deflected Subgradient Approach

```
Input: initial $w$, loss function $L$, initial $d$, $\beta$
for every example input $x$ do
    compute $o = wx$ and L
    using backprop compute $g = \nabla L$
    using (13) compute $\alpha$
    using (12) compute $d$
    using (14) compute $\eta$
    using (15) update $w$
```

2.3.1 What to expect from the algorithm

Due to the non-differentiability and non-convexity of my objective function, in theory I would expect a subgradient method to perform better than the heavy ball approach. This is certanly true in general for non-differentiable problems, but we are in a limit case due to the L2 regularization and the almost-convexity of the objective function. Anyway, with the right parameters, the algorithm is expected to converge to a minimum, which may be local or global. As in [**1**] a subgradient method requires $\Theta(\frac{1}{\epsilon^2})$ iterations to solve up to absolute error ϵ, which means that it is not practical for attaining any more than a modest accuracy. Besides, the complexity is independent of the size n of the problem. Therefore, SM may be promising for very-large-scale problems where a high accuracy is not necessary, whereas a short running time is a primary concern.

2.3.2 Implementation Overview

The following are the main implementation choices made for this algorithm. The algorithm is implemented into a class `a2` inheriting from class `m1`. `a1` consists of a main method `learning` which calls the others: `compute_alpha`, compute_deflection, compute_stepsize, update_weigths and backpropagation. main1.py calls the learning method of class a2 if specified by command line. The parameters used by the algorithm (k and β) are specified in a text file named "config_a2.txt" along with some model parameters.

2.3.3 Considerations and Results

We know that a fixed small stepsize leads to convergence, but is most often inefficient in practice. For this reason, I have chosen a rule similar to the *diminishing/square summable stepsize*, which maintains simplicity but enhances slightly faster convergence. The use of a convex combination in (12) is crucial, because it ensures that d_i is always an approximate subgradient of the loss function. The adding of deflection allows to face the ”zig-zagging” behaviour whereby $g_{i+1} \approx -g_i$, so that two ”reasonably long” steps combined make an ”unfeasibly short” one. In Figure 4 we can see the value of objective function varying with the number of epochs, using the the 5-th configuration in Table 1. In this case as well, the convergence is sublinear, more markedly than in Figure

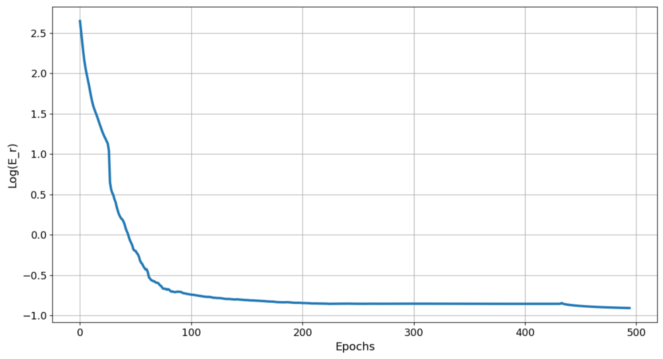

Figure 4: objcetive function varying with the number of epochs

2. This is consistent with the fact that since the objective function is almost convex, it is expected that a1 performs better than a2. This is because a2 is designed for scenarios where the lack of differentiability is significant enough to cause problems for algorithms like a1 that assume smoothness.

- *Config1:* $\beta = 0.003$, $k = 1500$
- *Config2:* $\beta = 0.001$, $k = 1500$
- *Config3:* $\beta = 0.003$, $k = 2000$

As we will see in 2.4, we are assuming that all algorithms converge to the same minimum. From the Figure 5, we can see that in each case, convergence is sublinear; there aren't significant performance differences, but the issue might be the speed of convergence.

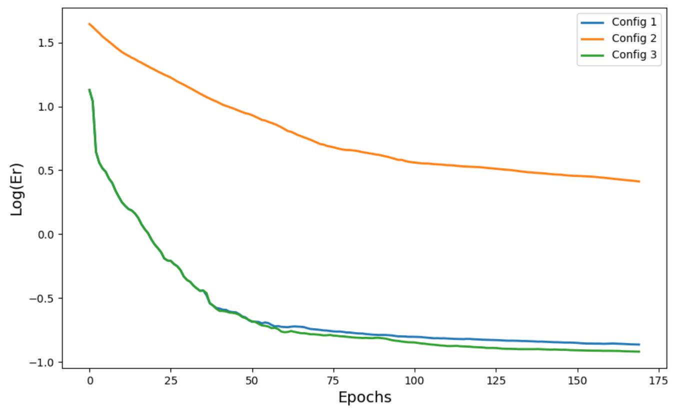

Figure 5: omparison between different parameters configurations

2.4 Experiments

Naming f the objective function MSE with regularization, in this section, we attempt to present and compare the results of the two algorithms discussed so far. The notation f^* specifies the minimum value reached by a specific configuration of the problem. As mentioned the objective function is not convex. For this reason, using the same notation of precious sections, in Table 1 I try to understand if the methods used were approaching the same optimal value or different ones, starting from the same initial point, creared with `np.random.uniform`. At first

Optimizer	Stepsize	Momentum	β	k	f^*
A1	0.0001	0.00001	-	-	0.28429
A1	0.001	0.0001	-	-	0.33832
A1	0.001	0.0005	-	-	0.24297
A1	0.00015	0.000015	-	-	0.25031
A2	DSS	-	0.003	1500	0.24012
A2	DSS	-	0.005	15000	0.17307
A2	DSS	-	0.001	1500	0.37842
A2	DSS	-	0.003	2000	0.23325

Table 1: Comparison of minima on 1500 iterations

glance, it seems that the various f^* values represent the same minimum, as they are small and similar. However, to evaluate this more accurately, this ratio was calculated:

$$\frac{\|w_i - \text{mean}_j\, w_j\|}{\|\max_j w_j\|}$$

And the obtained relative distances related to the data in the Table 1 are:

$$[0.56, 0.25, 0.18, 0.27, 0.29, 0.40, 0.21, 0.29]$$

We were interested in understanding whether the minimum reached by the two algorithms and some of their configurations was the same. We can reasonably assume it is, based on the calculated distances. This is reasonable, considering that we are assuming the objective function is not too non-convex. Therefore, if we are interested in determining the best parameter configuration for the algorithms, we should simply evaluate which one reaches the minimum the fastest. We can further validate our assumption regarding convexity by evaluating the minimum value reached in 200 epochs, for example, by algorithm A1 with the second parameter configuration in Table 1, using for each execution a different random initial point (`np.random.uniform` in $[-2, 2]$).

ex 1	0.509927
ex 2	0.410612
ex 3	0.554361
ex 4	0.489834
ex 5	0.417752

Table 2: Comparison of minima with different initial point

We can also conclude in this case that, despite the different initial point, the algorithm is converging to the same minimum value. Regarding the performance of the two algorithms on the problem, they do not seem to have significant differences, they both exhibit sublinear convergence and reach the same minimum with an acceptable number of iterations. Yet, it appears that a1 requires more iterations to converge, but it seems to continue decreasing in the tail, whereas a2 eventually stops minimizing.

3 Model 2: L2 Linear Regression

The second model aimed to solve a linear regression problem using *least squares*. So, my objective was to find $\overline{\mathbf{x}} \in \mathbb{R}^{nxk}$ s.t.

$$\overline{\mathbf{x}} = \text{argmin}_{\mathbf{x} \in \mathbb{R}^{nxk}} ||A\mathbf{x} - \mathbf{y}||_2$$

Where $A \in \mathbb{R}^{mxn}$ and $\mathbf{y} \in \mathbb{R}^{mxk}$. We can think of it as finding the vector closest to $\mathbf{y}$ in the hyperplane $Im(A)$.

3.1 Algorithm 3: QR solver

Among the direct linear least squares solvers, I chose the *QR factorization method* to solve least square problems.

Theorem. *For every $A \in \mathbb{R}^{mxn}$, there exists $Q \in \mathbb{R}^{mxm}$ orthogonal, R upper triangular s.t. $A = QR$.*

*Proof in Theorem 7.1 of [**3**].*
To compute the QR decomposition, I used the *Householder Reflectors* method.

Lemma. *$\forall$ v in $\mathbb{R}^m$ the matrix $H = I - \frac{2}{\|v\|^2}vv^T$ is orthogonal and symmetric.*

Proof. (symmetry)

$$H^T = I^T - \frac{2}{\|v\|^2}(vv^T)^T = I - \frac{2}{\|v\|^2}vv^T = H$$

(orthogonality)

$$H^T H = H^2 = (I - \frac{2}{\|v\|^2}vv^T)(I - \frac{2}{\|v\|^2}vv^T) = I - \frac{2}{\|v\|^2}vv^T) - \frac{2}{\|v\|^2}vv^T +$$

$$+\frac{4}{\|v\|^4}vv^T vv^T = I - \frac{4}{\|v\|^2}vv^T + \frac{4}{\|v\|^2}vv^T = I$$

□

Lemma. *Let x,y $\in \mathbb{R}^m$ s.t. $\|x\| = \|y\|$. If $v = x - y \Rightarrow H = I - \frac{2}{\|v\|^2}vv^T$ is s.t. $Hx = y$.*

Proof. $(I - \frac{2}{\|v\|^2}vv^T)x = y \Leftrightarrow (I - 2\frac{2vv^T}{2\|x\|^2 - 2x^T y})x = y \Leftrightarrow x - \frac{vv^T x}{\|x\|^2 - x^T y} = y$
$x - \frac{v(\|x\|^2 - y^T x)}{\|x\|^2 - x^T y} = y \Leftrightarrow x - (x - y) = y$ □

The vector $y = Hx$ can be computed from v and x with $O(m)$ operations. In our case, with A being the matrix of inputs and $\mathbf{y}$ being that of outputs, it is assumed that $m >> n$ (and indeed, this is the case in the example I will use in the implementation), so we will have a thin matrix. Using the QR facrorization, the problem will be:

$$||A\mathbf{x} - \mathbf{y}||_2 = ||QR\mathbf{x} - \mathbf{y}||_2 = ||Q^T(QR\mathbf{x} - \mathbf{y})||_2 =$$

Where the last equality stems from the fact that Q is ortogonal than it preserves the norm.

$$= \left\|R\mathbf{x} - Q^T\mathbf{y}\right\|_2 = \left\| \begin{bmatrix} R_0\mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} Q_0^T\mathbf{y} \\ Q_c^T\mathbf{y} \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} R_0\mathbf{x} - Q_0^T\mathbf{y} \\ -Q_c^T\mathbf{y} \end{bmatrix} \right\|_2$$

It is possible to notice that $-Q_c^T\mathbf{y}$ does not depend on $\mathbf{x}$. Therefore, the objective can be reduced to minimizing the upper part. $R_0\mathbf{x} - Q_0^T\mathbf{y}$ can be zero if and only if R_0 is invertible.

Theorem. *$\forall$ matrix A, the matrix R_0 given by its QR factorization is invertible iff A has full column rank.*

Proof. A has full column rank $\iff$ A^TA is invertible, but

$$A^TA = (QR)^TQR = R^TQ^TQR = R^TR = \begin{bmatrix} R_0^T & 0 \end{bmatrix} \begin{bmatrix} R_0 \\ 0 \end{bmatrix} = R_0^TR_0$$

Finally, the product of two square matrices is invertible iff they are both invertible. □

Then it is proved:

Lemma. *Let $A \in \mathbb{R}^{mxn}$ be a tall thin matrix with full column rank. Then the solution of the LS problem* $\min \left\|A\mathbf{x} - \mathbf{y}\right\|_2$ *is obtained with* $\mathbf{x} = R_0^{-1}(Q_0^T\mathbf{y})$ *and the value of the minimum is* $\left\|Q_c^T\mathbf{y}\right|_2$.

3.1.1 Implementation Overview

To address the problem presented in the dataset `ML-CUP23-TR.csv`, which contains approximately 1000 input data points, each with 10 features, I have chosen to use the *thin QR* factorization (also known as reduced QR factorization). This method is particularly suitable for rectangular matrices where the number of rows (1000) is significantly greater than the number of columns (10), allowing for a more efficient decomposition $A = QR$ compared to the full QR factorization. The thin QR factorization yields a matrix Q of dimensions mn (in this case $1000x10$) with orthonormal columns, and a matrix R of dimensions nn (10 x 10) that is upper triangular. Additionally, the *Huge optimization* was applied by leveraging the structure of the Householder matrix H. This approach avoids explicitly computing the product between matrices, effectively reducing the computational cost from cubic to quadratic (computing $R \leftarrow R - 2uu^TR$ instead of $R \leftarrow H$ and the same rispectively for Q). It is computationally much more convenient in thin cases. The algorithm computationally costs ($O(mn^2)$ from lecture 10 in [**3**]). The matrix corresponding to the dataset used has full column rank, So I was able to calculate $\mathbf{x}$ as described previously. I created a class `linear regression` that implements *model 2* for linear regression with LS, and class `a3` that implements an algorithm for LS problem via QR factorization.

Algorithm 3: QR thin solver for LS

```
Input: $A$,$x$
(m,n) ← shape(A)
R ← A
Q0 ← eye(m,n)
for $i$=0:$n$ do
    update $R$
    update $Q$
compute $x = Q_0^T y$
solve $R_0 x = Q_0^T y$
Output: x
```

3.1.2 Considerations and Results

Since QR decomposition is *backward stable* (the computed Q, R are the exact result of $qr(A + \Delta A)$ where $\|\Delta A\| \leq O(u)\,\|A\|$) and householder transformations are too, the total algorithm to solve LS problem with QR factorization is backward stable and we can say it "a priori"; so Algorithm 3 only have intrinsic representation errors. Since the problem is backward stable, I know that $\frac{\|\tilde{y}-y\|}{|y\|} \leq k_{rel}(f,x) \cdot O(u)$. I calculated $k_{rel}(f,x) = 7.912$, so we can affirm that the error in the solution is $\leq 7.912 \cdot 10^{-16}$. Moreover how much my QR factorization differs from the original matrix A, in terms of relative error, $\frac{\|A-QR\|}{\|A\|} = 1.09 \cdot 10^{-15}$, and this is very close to floating order. Moreover I computed relative residuals $\frac{\|Ax-y\|}{\|y\|} = 0.24325720795631878$ and compared with residuals from `np.linalg.lstsq`, these two values are the same up to the 18th decimal place. In order to verify if this implementation of thin QR decomposition using Householder reflections showed a linear growth in execution time as a function of the number of rows m of the matrix, as theoretically expected, Figure 6 depicts the graph of this growth. It is observed that, apart from some fluctuations, the execution time growth is regular and follows an increasing trend consistent with the expected linear growth.

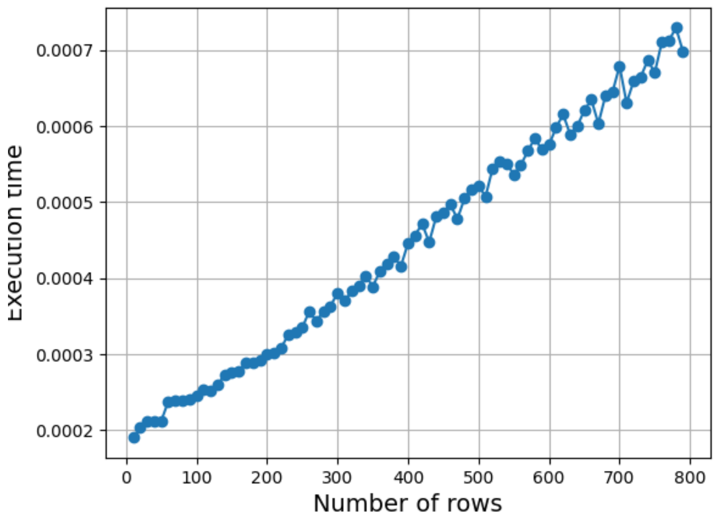

Figure 6: Thin QR Factorization Execution Time vs. Number of Rows

4 Bibliography

References

[1] Antonio Frangioni, Bernard Gendron, Enrico Gorgone. *On the computational efficiency of subgradient methods: a case study with Lagrangian bounds*, Math. Prog. Comp. (2017) 9:573–604 DOI 10.1007/s12532-017-0120-7.

[2] Giacomo d'Antonio, Antonio Frangioni *Convergence Analysis of Deflected Conditional Approximate Subgradient Methods*, SIAM J. OPTIM. Vol. 20, No. 1, pp. 357–386

[3] Lloyd N. Trefethen, David Bau *III Numerical linear algebra 1997* siam

[4] Ethem Alpaydın *Introduction to Machine Learning, Second Edition* The MIT Press Cambridge, Massachusetts London, England

[5] Euhanna Ghadimi, Hamid Reza Feyzmahdavian, and Mikael Johansson *Global convergence of the Heavy-ball method for convex optimization* 2015 European Control Conference (ECC) July 15-17, 2015. Linz, Austria

[6] Sébastien Bubeck *Convex Optimization: Algorithms and Complexity* Foundations and Trends in Machine Learning Vol. 8, No. 3-4 (2015) 231–358 c 2015 S. Bubeck