Optimization for Data Science

Project report

Gori Gabriele, Piccoli Davide

A.Y. 2023/2024

1. Introduction

Neural networks are fundamental tools in the field of machine learning, because of their ability to tackle complex tasks effectively. Despite their popularity, they are often referred to as ”black box models” due to their intricate internal workings.
A neural network is essentially defined by the function $y = f(x; \theta)$, where x represents the input data and y denotes the output, influenced by adjustable parameters θ optimized through the minimization of the error defined by a loss function.
The choice of the optimization algorithm significantly influences how these parameters θ evolve through the optimization process. This project explores a comparison between the momentum descent approach (Heavy-Ball) and the Proximal Bundle method, which is behind the Cutting Plane Model.
We'll start by observing the neural network's functioning and its associated loss function. Then, we'll examine the mechanics of the two algorithms and their theoretical foundations. Finally, we'll conduct an experimental analysis and get insights from the outcomes.

2. Neural Network

2.1 Structure

The structure of our neural network is described more deeply in section 6.1. In general, all the hidden layers and the output layer exploit the *tanh* activation function.
The loss function is the MSE and a regularization of type L1 makes sure that an addendum corresponding to the product between a penalty term λ and the sum of the absolute values of all the weights is added to the MSE formula (see 2.7).

2.2 Forward propagation

The first phase regards the forward propagation. Here an input is introduced to the model and goes through the hidden layers. Within this process, each input is multiplied by the weights and summed to the biases it meets while moving from a layer to the next one. Thus, the value of each neuron in the hidden layers is given by:

$$z_i = b_i + \sum_i x_i w_{ij} \tag{2.1}$$

where the first term represents the bias of that node, while the second term is the sum of all the products between any input from the previous layer and the weights connecting that layer with the node considered.
In our case, instead of considering one node at a time for every layer, we will replace this formula by the following matrix product for the entire layer *l*:

$$Z^l = B^l + X^l W^l \tag{2.2}$$

where the terms are replaced by a vector and a matrix respectively.
The layer's output then passes through the activation function attached to that layer, so that the resulting value would be:

$$A^l = \sigma(Z^l) \tag{2.3}$$

As explained previously, we have chosen the *tanh* (hyperbolic tangent function) for the activation of hidden and output layers. The tanh function is given by:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.4}$$

The same procedure applies to every hidden layer $l \in [0, 1, ..., L]$, until the output layer is reached. Thus, the final output of the net is:

$$Y = \sigma(Z^L) \tag{2.5}$$

Considering the above-mentioned notation, a scaled representation of our neural network can be observed in the image below.

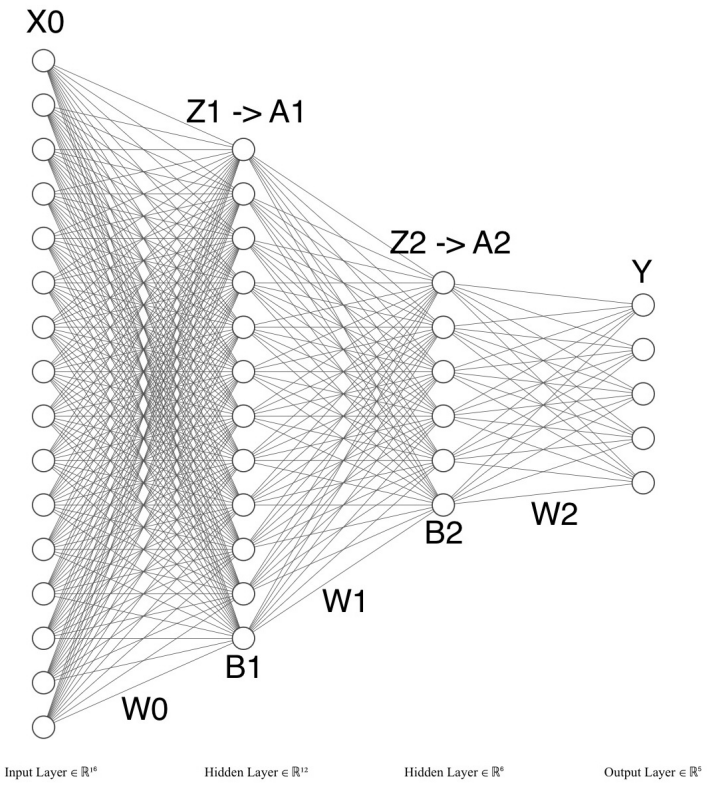

Figure 1: Scaled representation of the neural network, according to the notation presented before. Inputs and outputs of every layer, as well as their weights and biases, are treated as matrices.

2.3 Backpropagation

Our cost function is *MSE*:

$$E = \sum_{i=1}^{N} (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 \tag{2.6}$$

where $\hat{\boldsymbol{y}}_i$ is the actual value we want to predict. Our assignment required an **L1 regularization** term, which is a penalization term. The cost function then changes as follows:

$$E = \sum_{i=1}^{N} (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 + \lambda \|w\|_1 \tag{2.7}$$

Starting from the error, we want to implement the **backpropagation** scheme. The cost function is what we want to optimize, taking steps towards the ideal direction. In order to make our model actually learn something, we need the derivative of the cost function with respect to all the model's parameters. To achieve that result, the standard backpropagation scheme exploits the chain rule, starting from the last layer and all the way back to the first one.
Since the *MSE* function is just composed by two terms that add up, we can treat them separately. The derivative of the actual error term of the *MSE* is:

$$E' = \frac{2}{N}(\boldsymbol{Y} - \hat{\boldsymbol{Y}}) \tag{2.8}$$

The L1 regularization term (which includes an absolute value), instead, is not differentiable. Thus, here we can exploit the concept of subgradient that we will examine in Chapter 4.. The subgradient of the L1 term can be expressed as:

$$\partial_W(\lambda \|W\|_1) = \begin{cases} \lambda & if W > 0 \\ -\lambda & if W < 0 \\ [-\lambda, \lambda] & if W = 0 \end{cases}.$$

For convenience, in the actual implementation we consider the derivative to be 0 in the case $W = 0$. Starting from the last layer *l*, the derivative with respect to the Weights W^l is:

$$\frac{\partial \tilde{E}}{\partial W^l} = \frac{\partial \tilde{E}}{\partial Y} \frac{\partial Y}{\partial Z^l} \frac{\partial Z^l}{\partial W^l} + \lambda \frac{\partial \|W\|_1}{\partial W^l} \tag{2.9}$$

while the derivative with respect to B^l:

$$\frac{\partial \tilde{E}}{\partial B^l} = \frac{\partial \tilde{E}}{\partial Y} \frac{\partial Y}{\partial Z^l} \frac{\partial Z^l}{\partial B^l} \tag{2.10}$$

Hence, if we want to back-propagate further we just need to back-propagate the $\frac{\partial \tilde{E}}{\partial Y}\frac{\partial Y}{\partial Z^l} = \frac{\partial \tilde{E}}{\partial Z^2}$ term. So, the backpropagation scheme for the layer *l-1* is characterized by:

$$\frac{\partial \tilde{E}}{\partial W^{l-1}} = \frac{\partial \tilde{E}}{\partial Z^l}\frac{\partial Z^l}{\partial A^{l-1}}\frac{\partial A^{l-1}}{\partial Z^{l-1}}\frac{\partial Z^{l-1}}{\partial W^{l-1}} + \lambda\frac{\partial \|W\|_1}{\partial W^{l-1}} \tag{2.11}$$

$$\frac{\partial \tilde{E}}{\partial B^{l-1}} = \frac{\partial \tilde{E}}{\partial Z^l}\frac{\partial Z^l}{\partial A^{l-1}}\frac{\partial A^{l-1}}{\partial Z^{l-1}}\frac{\partial Z^{l-1}}{\partial B^{l-1}} \tag{2.12}$$

Exploiting this scheme, we can compute all the derivatives we need for the learning process of our Neural Network, covering all parameters.

3. Heavy ball method

The heavy ball method (or momentum descent) aims at improving the performance of a gradient descent (GD) approach. GD updates the parameters as follows:

$$W_{t+1} = W_t + \text{stepsize} \cdot \nabla E(W_t)$$

GD is the simplest approach in machine learning. It exploits the first order information to compute the stepsize (which is obviously a value $\in [0, 1]$).
On the other hand, the heavy ball method updates the parameters through the following procedure:

$$W_{t+1} = W_t + \Delta W_{t+1}$$

where:

$$\Delta W_{t+1} = \beta \cdot \Delta W_t - \text{stepsize} \cdot \nabla E(W_t)$$

where β is the momentum term. The step is basically given by the negative gradient step plus another term which is the previous step scaled by the factor β. If the factor $\beta = 0$, the momentum term is not computed, so it returns to be a standard gradient descent algorithm. This method differentiates from the vanilla gradient descent by taking into account the history of the previous gradients and thus adding a sort of ”inertia” to the process of convergence. This feature ensures that the algorithm is robust when local minima or saddle points are encountered in the optimization process, thus its convergence is faster than the one observed with the standard gradient descent.

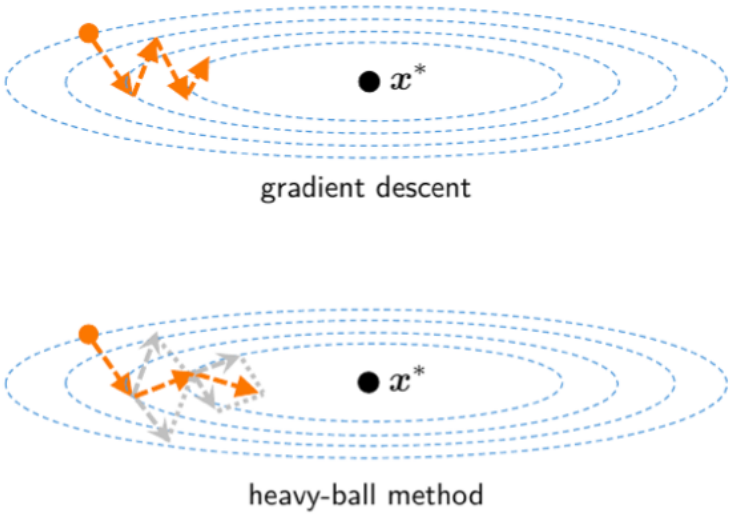

Figure 2: Comparison between convergence processes of the standard gradient descent and the gradient descent with momentum.[3]

3.1 Heavy Ball convergence

The convergence of the Heavy Ball method for non-convex functions is less straightforward than for strongly convex ones.

Theorem 1. *For f strongly τ-convex and L-smooth, with the optimum choice of parameters the Heavy Ball method converges as*

$$O\left(\frac{\sqrt{L}-\sqrt{\tau}}{\sqrt{L}+\sqrt{\tau}}\right)$$

This result is achieved when the step-size parameters are chosen optimally, as described in Polyak's analysis [5]. However, in the case of non-convex functions, such as those arising from our case (e.g., *tanh* activation functions in our neural network), the guarantee of this convergence rate no longer holds. Non-convexity in the objective function complicates the analysis. To handle this, we can turn to more general results for non-convex optimization. A suitable approach is given by the iPiano algorithm [4], which addresses the minimization of composite objective functions:

$$\min_{x\in\mathbb{R}^n} f(x) = h(x) + g(x)$$

where $h(x)$ is L-smooth but potentially non-convex (such as the Mean Squared Error, MSE, in our case), and $g(x)$ is convex but possibly non-smooth (such as the L_1 regularization term).

Theorem 2. *Given a function f(x) = h(x) + g(x), under the conditions that $h(x)$ has an L-Lipschitz gradient and $g(x)$ is lower semi-continuous, the iPiano algorithm has a convergence rate for the squared error*

$$O\left(\frac{1}{k}\right)$$

The conditions stated in Theorem 2 apply to our scenario: the MSE is differentiable with an L-Lipschitz gradient, ensuring smoothness, and the L_1 regularization is convex and lower semi-continuous due to the continuity of the absolute value function. Thus, while the Heavy Ball method may not guarantee the same optimal convergence in our non-convex case, the iPiano algorithm provides a more suitable framework for analyzing the convergence of our objective function, which combines both smooth and non-smooth components. Later on, this analysis will constitute a benchmark in order to compare the efficiency of this algorithm with the one resulting from Bundle methods.

4. Bundle methods

First of all, we need to define the **subgradient** we have already mentioned before. A vector *g* is a subgradient of a convex function $f(x)$ if:

$$f(z) \geq f(x) + \langle g, z - x \rangle \forall z \in Domain(f)$$

The problem is that for each point we have a lot of possible subgradients. Here it comes the subdifferential:

$\partial(f) = \{s \in \mathbb{R}^n$ s is a subgradient at x$\} \equiv$ subdifferential

$\partial(f) = \{\nabla f(x)\} \iff f$ differentiable at x

d is a descent direction $\iff \langle s, d \rangle < 0 \forall s \in \partial f(x)$. Non-differentiable optimization is orders of magnitude slower. If $f \notin C^1$ there can be many different subgradients. We assume that we have an *oracle* that is defined as:

Oracle: *at any point $x \in Domain(f)$, the output is $f(x), g \in \partial f(x)$ which is the first-order model at x.*

Bundle methods is a family of techniques which is primarily conceived for solving non-smooth optimization problems. These algorithms exploit the concept of **cutting planes**, a technique that approximates a non-linear function with a simpler one. This approach involves creating a piecewise linear function (”cutting plane”) that lies below the graph of the original function f at a specific point w'. This linear approximation gives an estimate of f at that particular point.

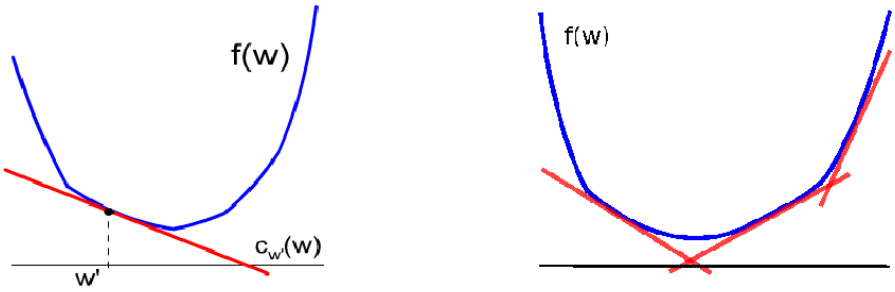

Figure 3: On the left: basic approximation of a function f by a single cutting plane at a point w'. On the right: a more accurate approximation considering multiple cutting planes of f.[1]

In other words, we suppose that in addition to our current iteration point x_k, we obtain some trial points $y_j \in \mathbb{R}^n$ and subgradients $\xi_j^T \in \partial f(y_j)$ for $j \in J_k$. Our model function (approximated function $\hat{f}_k(x)$) is defined by the cutting plane model:

$$\hat{f}_k(x) := \max_{j \in J_k} \{f(y_i) + \xi_j^T (x - y_j)\}$$

The optimization process exploits a bundle of subgradients which is used to update the parameters at each step. In other words, Bundle Methods incorporate information from past iterations into their models, allowing $\hat{f}(x)$ to capture the behaviour near x_k. At each iteration we want to solve a *master problem* to compute a d^*, which is the candidate step. In our code,

$$d^* = argmin\{f_B + \frac{\mu \|d\|^2}{2}\}$$

which can also be seen as:

$$d^* = argmin\{f_B + \frac{\mu \|x - \bar{x}\|^2}{2}\} \tag{4.13}$$

is the *stabilized* master problem. Essentially, we want to keep the next value close to our stability center $\bar{x}$ (our best x_k so far).
The pseudocode of our implementation for a Proximal Bundle Method (PBM) is the following:

PBM Proximal bundle method($f, subgrad_f, x, m1, \epsilon, \mu, max\_iter, max\_sub$):

B = [(x, f(x), subgrad_f(x))]
while $k \leq max\_iter$ **do**:
$d^* \longleftarrow argmin\{f_B + \frac{\mu \|d\|^2}{2}\}$
if $f(x + d^*) - f(x) \leq m1 \cdot [f_B(x + d^*) - f(x)]$ **then**
$x \longleftarrow x + d^*$; $\mu \searrow$
else
$\mu \nearrow$

end if
$B \longleftarrow B \cup \{(x + d^*, f(x + d^*), subgrad_f(x + d^*))\}$
if $len(B) > max\_sub$ **then**
pop first element in the bundle
end if
end while

The parameter $m1$ makes sure that we are doing a *serious enough* step, otherwise the algorithm just decreases μ in a null-step.
We wanted to make this code suitable for the learning of a Neural Network, primarily to manage learning properly but also to optimize running time. An automatic solver is exploited for the master problem, which is the `'CPLEX'` method (short for "IBM ILOG CPLEX" developed at IBM). In a previous version we also tested the open-source `'HiGHS'` optimizer, designed for solving large-scale linear programming (LP) and mixed-integer programming (MIP) problems; however, we definitely prefer`'CPLEX'` due to its robustness.
The pseudo-code highlights that the main goal is to minimize the quadratic model $\{f_B + \frac{\mu\|d\|^2}{2}\}$. This model has two components: $\{f_B\}$ which represents the function value at the new point $(x + d)$ and $\frac{\mu\|d\|^2}{2}$ which involves μ and $\|d\|^2$ in order to penalise large steps.
Our version of PBM is a limited-memory variant, since only a *max_sub* number of previous subgradients evaluations is exploited. The algorithm keeps only the last max_sub subgradients, in a first-in-first-out fashion. The parameter *max_sub* can be chosen by the user. Now it's important to clarify that the PBM version we want to test has no guarantee about its convergence, and performances on NNs will be discussed later on (Section 6.1). The important point in our opinion is that, even though our limited memory variant is not too sophisticated, we take advantage from the relevant decrease in terms of computational time, especially for Neural Networks (where a full-memory variant is too slow in practice).

4.1 Bundle methods convergence

Unfortunately, convergence study in our setting is not an easy task. We just report some theoretical results to provide some context. As Mateo Dìaz and Benjamin Grimmer state in a recent study [2], bundle methods are known to converge under some assumptions, but non-asymptotic analysis have been mostly evasive. Our problem also involves L1 norm and non-convexity, which add a lot of complexity in the convergence analysis.
We start considering the two components separately, since a comprehensive study is not available for our case. We firstly consider the MSE component of our cost function, which we already know that it is L-smooth. The convergence rate that Diaz and Grimmer found for such a (convex) scenario is:

$$O\left(\frac{L^3\|x_0 - x^*\|^2}{\mu^2\varepsilon}\right) \tag{4.14}$$

Thus, if our function was convex (but it is not) the convergence rate according to the authors would become:

$$O\left(\frac{L\|x_0 - x^*\|^2}{\varepsilon}\right) \tag{4.15}$$

We specify that the authors considered the rate for an ε-minimizer, such that: $f(x) - f^* \leq \varepsilon$.
The L1 component, instead, is not L-smooth. However, it can be proven, exploiting the triangle inequality, that the L1 norm is 1-Lipschitz continuous. Now we make use of the linearity of Lipschitz continuity, which states that *the sum of Lipschitz continuous functions is Lipschitz continuous*. Hence, our entire Cost function E is Lipschitz continuous. Moreover, the mentioned paper also considers the Hölder continuity and growth, defined as follows:

Hölder continuity: *a function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be Hölder continuous if there exist constants $C \geq 0$ and $0 < \alpha$ such that for all $x, y \in \mathbb{R}^n$, we have:*

$$|f(x) - f(y)| \leq C\|x - y\|^\alpha$$

where the constant α is called the Hölder exponent of the function. If $\alpha = 1$, the function is Lipschitz continuous. Lipschitz continuity is a stronger property than Hölder condition. Since the entire Cost function in our case is just Lipschitz continuous, then is straightforward to prove that $\alpha = 1$. This concept is strongly related to what in [2] is defined as:

Hölder growth: *a function $f : \mathbb{R}^n \to \mathbb{R}$ is said to satisfy Hölder growth if for all $x \in \mathbb{R}^n$, we have:*

$$f(x) - f^* \geq C * dist(x, X^*)^\alpha$$

where $dist(x, S) = inf_{y \in S}|x - y|$. This concept is crucial here because the authors distinguish between:

- $\alpha = 1$ (sharp growth, μ -SG)
- $\alpha = 2$ (quadratic growth, μ -QG)

In this case, assuming $\alpha = 1$ also for growth, the authors *would* provide a convergence rate (without μ tuning):

$$O\left(\frac{M^2}{\mu\varepsilon}\right) \tag{4.16}$$

The discussion for Hölder growth in our setting is omitted, since we want to remind that this is just a theoretical framework to give a rough idea of what a Bundle Method is capable of in terms of convergence. However, since the study only considers **convex** settings (which is not the case in our Neural Network), the aim of this section is just to report the main available results without the pretense of being exhaustive. Our empirical results will be discussed later on in Chapter 6..

5. Experiments on demonstrative functions

Before implementing our functions on our specific case, we wanted to check whether and how they could perform on simpler functions. This step would make it easier for us to detect and fix possible issues with our algorithms. Specifically, Rosenbrock's and Himmelblaus's functions were involved in this test.
A comparison of Proximal Bundle and Heavy Ball methods on Rosenbrock's function can be observed below. The Proximal Bundle Method has the learning rate equal to 0.001 and the stability term μ equal to 10 (adjusted adaptively). For what concerns the Heavy Ball Method, the learning rate is equal to 0.001 and the momentum is equal to 0.8. We used a *max_sub*=50 for BM, which is the maximum number of past subgradients to keep in memory.
The two comparisons are different in terms of starting points and number of iterations.

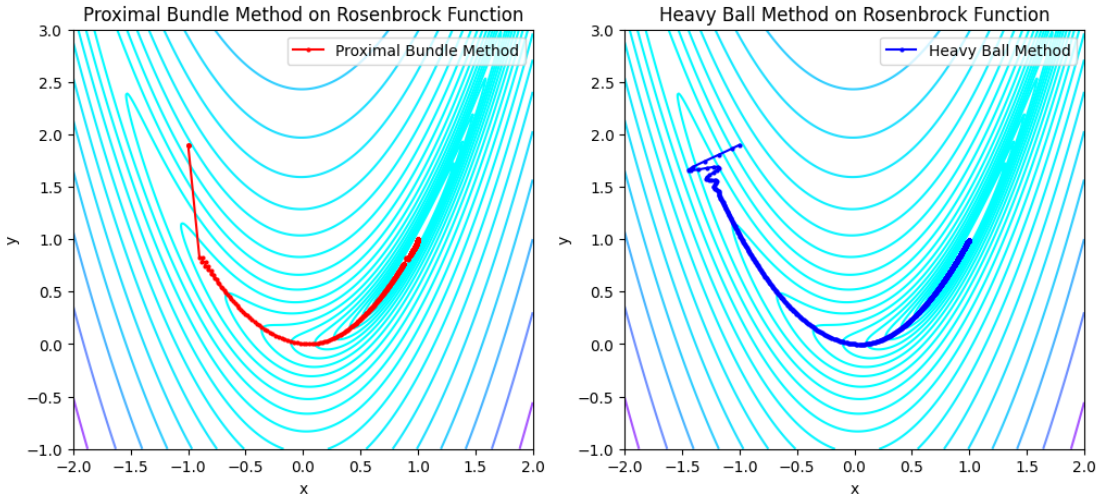

Figure 4: Comparison of Proximal Bundle Method and Heavy Ball Method on Rosenbrock Function. The starting point is x=-1.0, y=1.9 and the maximum number of iterations is 4000.

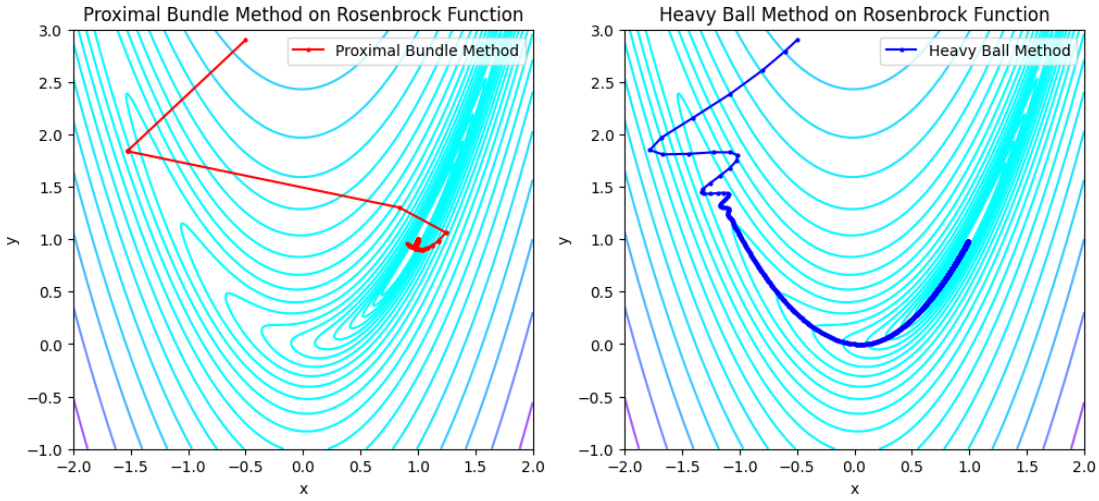

Figure 5: Comparison of Proximal Bundle Method and Heavy Ball Method on Rosenbrock Function. The starting point is x=-0.5, y=2.9 and the maximum number of iterations is 4000.

The previous comparison (with starting point x=-0.5, y=2.9) is also represented on a 3d-scale in the image below.

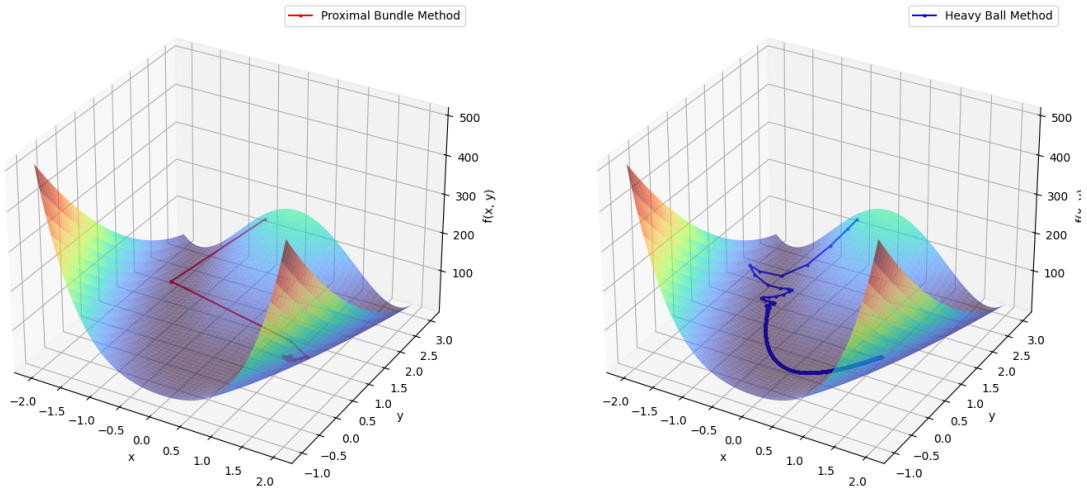

Figure 6: Comparison of Proximal Bundle Method and Heavy Ball Method on Rosenbrock Function on a 3d scale. The starting point is x=-0.5, y=2.9 and the maximum number of iterations is 2000.

The results related to Himmelblau's function are shown below and the parameters of the two algorithms are the same as before. The two examples below differ in the location of the starting point.

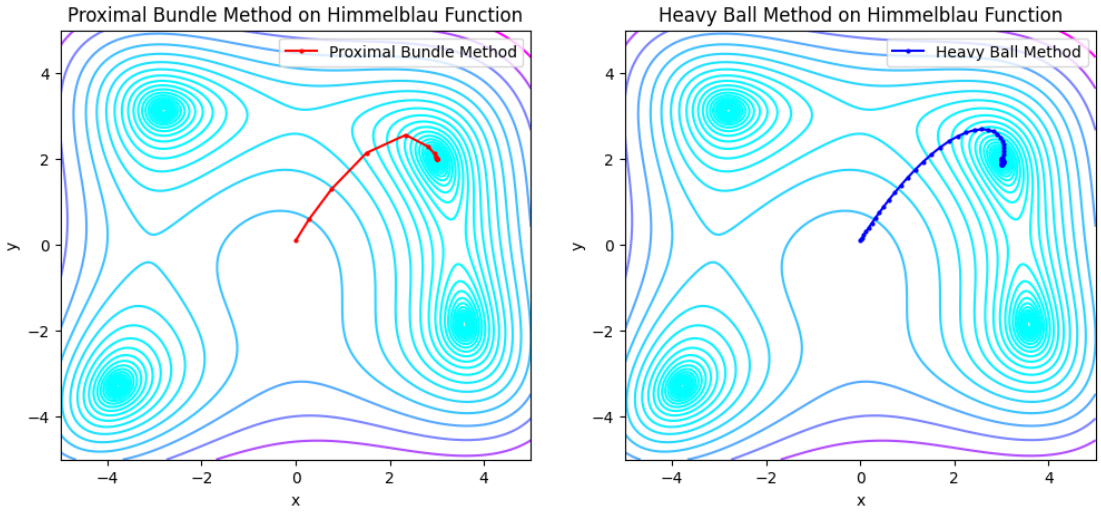

Figure 7: Comparison of Proximal Bundle Method and Heavy Ball Method on Himmelblau Function. The starting point is x=0.0, y=0.1. The maximum number of iterations is set to 2000.

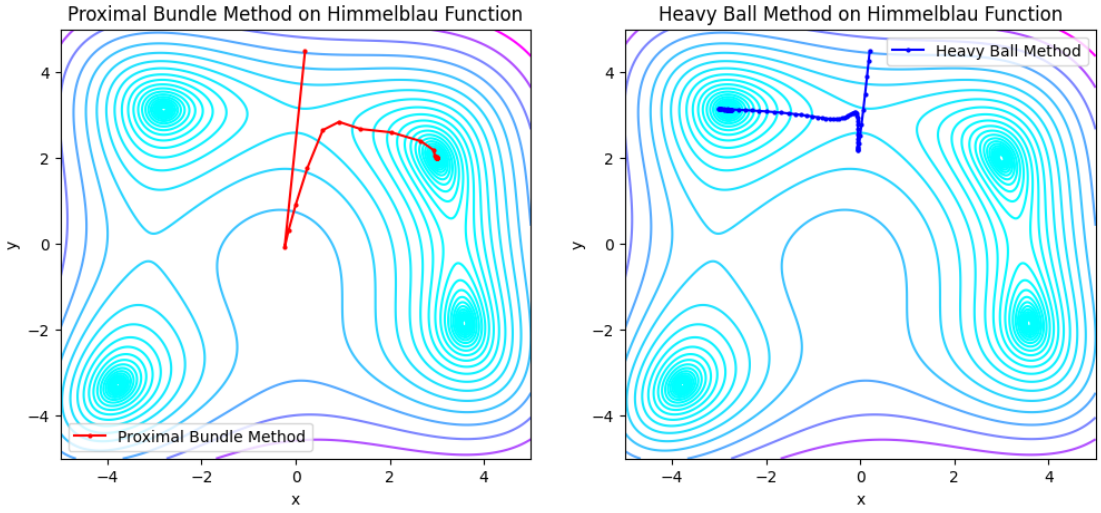

Figure 8: Comparison of Proximal Bundle Method and Heavy Ball Method on Himmelblau Function. The starting point is x=0.2, y=4.5. The maximum number of iterations is set to 4000.

Also in this case, the previous comparison (with starting point x=0.2, y=4.5) is then represented on a 3d-scale in the following image.

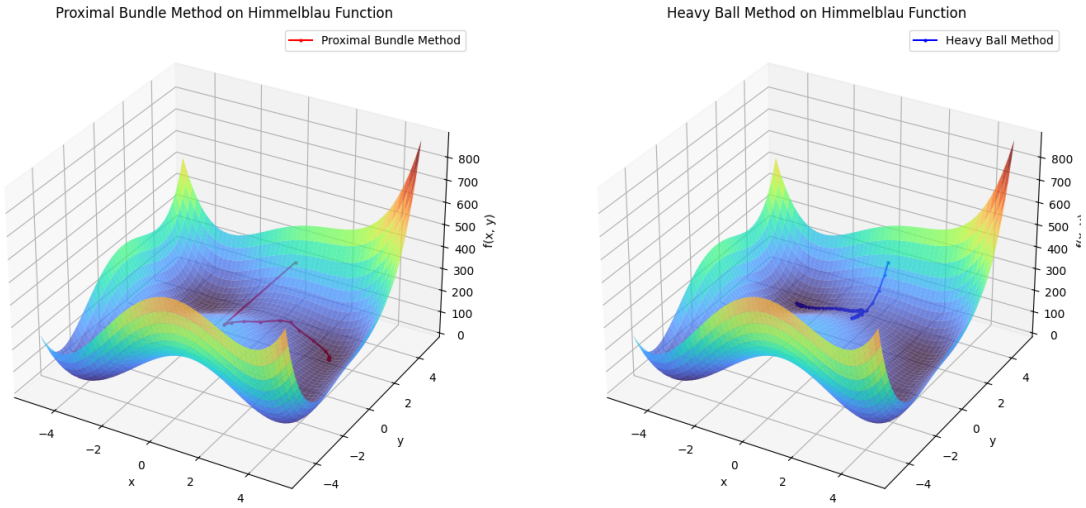

Figure 9: Comparison of Proximal Bundle Method and Heavy Ball Method on Himmelblau Function on a 3d scale. The starting point is x=0.2, y=4.5. The maximum number of iterations is set to 4000.

The experiments on these demonstrative functions proved that both methods are able to converge. In every case, they both were very close to the minimum point when the iterations threshold was met.
Figures 5 and 8 highlight that the Heavy Ball momentum term prevents its convergence to make such large oscillations as the one clearly visible on the Proximal Bundle Method paths. As a result, the Heavy Ball paths from the starting point to the minimum point look smoother compared to those realised by the Proximal Bundle Method.

6. Experiments on Neural Networks

At this point the two algorithms we previously designed are used to minimize the Loss (MSE + L1) in the backward propagation phase of our neural network implementation. There are two operations we perform at each epoch, which are related to weight updates and bias updates of each layer. We exploit the derivatives we discussed in Section (2.3) for both weights and biases.
As an attempt to test the behaviour of our Net implementation from scratch, our neural networks were first applied on the Iris dataset and then on the Wine dataset. The aim was to test results with well known yet simple datasets, to find a trade-off between computational time and effectiveness.

6.1 Random search methodology

This section reports the results for the random search we performed for HB and PBM, exploiting two different layouts:

- NN with one hidden layer of 10 neurons
- NN with two hidden layers of 10 neurons each

The activation function for all layers is the $tanh$.
The maximum number of epochs was set to 2000 for each model in the experiments to make the results comparable. Moreover, we noticed that our PBM implementation was slower than the HB implementation, so we believe this to be a good trade-off to test a lot of combinations of different hyperparameters for both algorithms.
Testing our Heavy Ball implementation, we noticed an effective and reliable behaviour: this is why we used an HB long run (10,000 epochs) as benchmark for all the experiments. These long runs provide an useful approximation of the minimum we aim to find in our tests. The best benchmark value we could obtain with HB is reported below the Tables.
In Figure (10) we report the performance results in terms of Loss (with L1) history on a logarithmic scale, just for visualization purposes. We notice that using HB the vast majority of *learning* is made within the first 200/300 epochs, while PBM shows a much rapid initial convergence. This advantage comes at a cost and we will discuss computational times later on (see tables in 6.2 and 6.3). After 500 epochs changes are very small for both algorithms, and the most important factor becomes the adaptive behaviour.

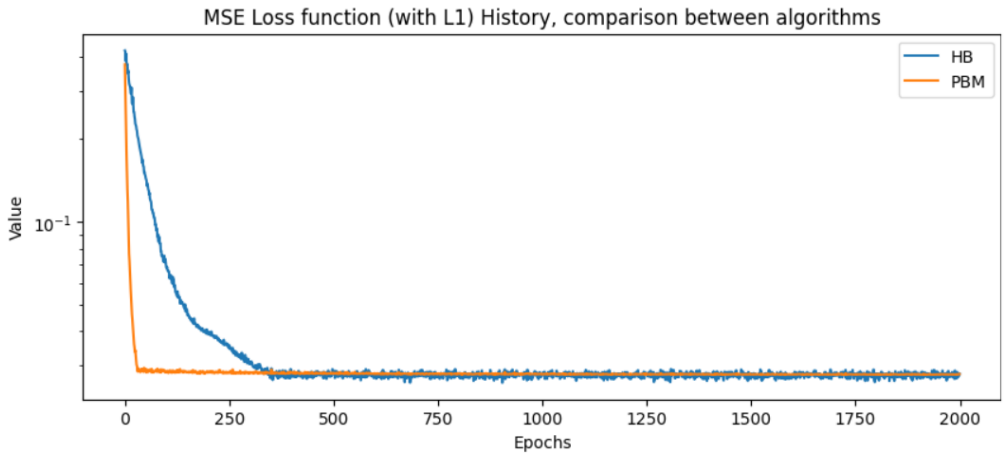

Figure 10: Heavy Ball and Proximal Bundle Method comparison (logarithmic scale) on Iris dataset.

First we performed a random search for Heavy Ball. The value λ for the L1 regularization was strictly set to 0.05 for every experiment from now on. We implemented a stopping

criterion which is the maximum number of epochs, set to 2000 (to avoid infinite runs, since we did many of them). Then for HB we added an adaptive learning rate which:

- ignores the first 5 iterations
- starting from the 6th iteration, checks whether the loss function value increased with respect to the previous iteration
- if the loss function increased, $learning\_rate * \rho$ with $0 < \rho < 1$ and ρ can be chosen arbitrarily.

In our tests the ρ was set to 0.9999. By that time, we also wanted to perform a random search for PBM with some random values for μ, max_sub and m_1 to see whether it could achieve similar performances or not. The stopping criteria are quite the same as in HB but, since there is no learning rate, we used the following criteria:

- ignores the first 5 iterations
- starting from the 6th iteration, checks whether the loss function value increased with respect to the previous iteration
- if the loss function increased, the parameter μ becomes $\mu * (1 + \rho)$ with ρ chosen arbitrarily (thus μ is increased too)
- if the loss function decreased, the parameter μ becomes $\mu * (1 - \rho)$ with ρ chosen arbitrarily (thus μ is decreased too)

The first 5 steps are skipped to let the PBM algorithm find a ”good” direction, since sometimes it is quite unstable within the first iterations. With regard to adaptivity, we record that especially for high-momentum terms the HB algorithm keeps bouncing around the solution, so we wanted to limit this behavior. The tables in the next sections recap the results of the random search experiments we performed. Several parameters combinations were tested for that, always keeping L1 lambda at 0.05; we finally took the best result for that particular problem, testing all the combinations of layers and Iris/Wine datasets. For PBM tests, the same appropriate benchmark was taken as goal. 'Relative gap' is then computed as:

$$Relative\_gap = \frac{Loss_{run} - Loss^*}{Loss^*}$$

where $Loss^*$ is the benchmark value we are considering. We would like to remind that $Loss$ is always MSE+L1 term in our tests. Running 'Time' is expressed in seconds.

6.2 Heavy Ball method results (HB)

In this subsection we discuss the results obtained with the Heavy Ball method on both datasets. Below every table, we report the Benchmark for that particular problem.

6.2.1 Iris dataset

Table 1: Iris Dataset with 1 hidden layer

Experiment	Result	Relative Gap	Epochs	Time (s)
α=0.03, momentum=0.5	0.027248	0.001972	2000	106.55
α=0.01, momentum=0.9	0.027453	0.009518	2000	109.20
α=0.03, momentum=0.1	0.027541	0.012740	2000	102.10
α=0.01, momentum=0.3	0.027693	0.018367	2000	90.61
α=0.01, momentum=0.9	0.027796	0.022133	2000	81.88
α=0.015, momentum=0.7	0.027883	0.025315	2000	202.91

Heavy Ball results on Iris Dataset. Benchmark with 10,000 HB epochs = 0.0271943584084

Table 2: Iris Dataset with 2 hidden layers (HB)

Experiment	Result	Relative Gap	Epochs	Time (s)
α=0.03, momentum=0.9	0.026799	0.053416	2000	104.13
α=0.01, momentum=0.5	0.027511	0.081413	2000	85.16
α=0.01, momentum=0.9	0.027598	0.084817	2000	119.47
α=0.01, momentum=0.9	0.028175	0.107514	2000	112.44
α=0.01, momentum=0.3	0.028366	0.115025	2000	93.27
α=0.00015, momentum=0.7	0.028425	0.117339	2000	89.68

Heavy Ball results on Iris Dataset. This version exploits two hidden layers of 10 neurons. Benchmark with 10,000 HB epochs = 0.0254398286342

6.2.2 Wine dataset

Table 3: Wine Dataset with 1 hidden layer (HB)

Experiment	Result	Relative Gap	Epochs	Time (s)
α=0.03, momentum=0.1	0.016710	0.005210	2000	169.90
α=0.015, momentum=0.7	0.016833	0.012499	2000	65.92
α=0.01, momentum=0.3	0.016960	0.019915	2000	106.43
α=0.015, momentum=0.4	0.017388	0.044028	2000	160.33
α=0.07, momentum=0.1	0.018834	0.117422	2000	168.15
α=0.03, momentum=0.6	0.020230	0.178300	2000	63.60

Heavy Ball results on Wine Dataset. Benchmark with 10,000 HB epochs = 0.016622.

Table 4: Wine Dataset with 2 hidden layers (HB)

Experiment	Result	Gap	Epochs	Time (s)
α=0.01, momentum=0.1	0.024626	0.000556	2000	66.35
α=0.07, momentum=0.1	0.025311	0.028385	2000	140.25
α=0.005, momentum=0.3	0.026484	0.076096	2000	62.65
α=0.003, momentum=0.2	0.027767	0.128219	2000	91.36
α=0.001, momentum=0.5	0.028330	0.151089	2000	87.31
α=0.015, momentum=0.2	0.028778	0.169267	2000	110.82

Heavy Ball results on Wine Dataset. This version exploits two hidden layers of 10 neurons. Benchmark with 10,000 HB epochs = 0.024612.

Heavy Ball can reach a good gap with the benchmark in both the datasets. We report the most interesting runs, showing that the algorithm is capable of reaching the minimum we found previously with 10,000 epochs. Experiments with 2 layers show more variance in results, probably due to the increased complexity of the problem. In our opinion, it is important to keep in mind that there is a non-negligible amount of randomness in NNs initializations, so a specific run can be lucky in a certain way (this is also why we took different HB runs to find a benchmark). The important thing we consider is that our Net is capable of reaching the same goal as in the benchmarks best run, even though sometimes the non-convex environment takes the algorithm in a bad direction.

6.3 Proximal Bundle method results

In this subsection we discuss the results obtained with the Proximal Bundle method on both datasets. Since, as we discussed previously, Heavy Ball is our most reliable method, here we report the same benchmark as in the previous experiments (for all layers and datasets combination, respectively).
Considering the different nature of this algorithm with respect to Heavy Ball, here the subproblem running times are shown too.

6.3.1 Iris dataset

Table 5: Iris Dataset with 1 hidden layer (PBM)

Experiment	Result	Relative Gap	Time (s)	Epochs
μ=0.05, max_sub=20, m_1=0.001	0.027881	0.025236	953/565	2000
μ=0.05, max_sub=20, m_1=0.01	0.027886	0.025446	751/459	2000
μ=0.1, max_sub=20, m_1=0.005	0,028007	0.029888	732/453	2000
μ=0.2, max_sub=30, m_1=0.005	0,028045	0.031285	1327/736	2000
μ=0.2, max_sub=10, m_1=0.03	0.028102	0.033364	1527/916	2000
μ=0.1, max_sub=20, m_1=0.01	0.028482	0.033905	809/490	2000

Proximal Bundle Method results on Iris Dataset. The column 'Time' reports, in seconds, both the overall running time (first value) and the subproblem's running time.
Benchmark with 10,000 HB epochs = 0.0271943584084

Table 6: Iris Dataset with 2 hidden layers (PBM)

Experiment	Result	Relative Gap	Time (s)	Epochs
μ=0.05, max_sub=20, m_1=0.01	0.027963	0.099192	712/456	2000
μ=0.05, max_sub=20, m_1=0.001	0.028067	0.103263	1627/628	2000
μ=0.1, max_sub=20, m_1=0.01	0.028088	0.104079	1021/622	2000
μ=0.1, max_sub=20, m_1=0.005	0,028169	0.107294	975/603	2000
μ=0.2, max_sub=30, m_1=0.005	0.028306	0.112679	4311/2689	2000
μ=0.2, max_sub=10, m_1=0.01	0.028336	0.113824	2938/1858	2000

Proximal Bundle Method results on Iris Dataset. The column 'Time' reports, in seconds, both the overall running time (first value) and the subproblem's running time. This version exploits two hidden layers of 10 neurons. Benchmark with 10,000 HB epochs = 0.0254398286342

6.3.2 Wine dataset

Table 7: Wine Dataset with 1 hidden layer (PBM)

Experiment	Result	Relative Gap	Time (s)	Epochs
μ=0.2, max_sub=20, m_1=0.07	0.018535	0.114745	8231/6970	2000
μ=0.1, max_sub=10, m_1=0.001	0.018889	0.136013	1596/1103	2000
μ=0.1, max_sub=5, m_1=0.005	0.020085	0.207953	2964/2428	2000
μ=0.05, max_sub=5, m_1=0.1	0.020094	0.208511	1655/1244	2000
μ=0.1, max_sub=20, m_1=0.07	0.020801	0.250996	6578/4973	2000
μ=0.2, max_sub=30, m_1=0.01	0.020227	0.282199	7048/4803	2000

Proximal Bundle Method results on Wine Dataset. The column 'Time' reports, in seconds, both the overall running time (first value) and the subproblem's running time. Benchmark with 10,000 HB epochs = 0.016622.

Table 8: Wine Dataset with 2 hidden layers (PBM)

Experiment	Result	Relative Gap	Time (s)	Epochs
μ=0.3, max_sub=10, m_1=0.01	0.026073	0.059361	1746/1196	2000
μ=0.05, max_sub=15, m_1=0.0001	0.026583	0.078254	2260/1507	2000
μ=0.01, max_sub=20, m_1=0.07	0.027300	0.109215	4602/2805	2000
μ=0.003, max_sub=10, m_1=0.005	0.027952	0.135706	5677/5001	2000
μ=0.08, max_sub=30, m_1=0.06	0.027978	0.136762	4988/3149	2000
μ=0.02, max_sub=20, m_1=0.07	0.029574	0.201608	3921/2578	2000

Proximal Bundle Method results on the Wine Dataset. The column 'Time' reports the overall running time (first value) and the subproblem's running time. This version uses two hidden layers of 10 neurons. Benchmark with 10,000 HB epochs = 0.024612.

As we can see from the Tables, even though the algorithm is able to reach good relative gaps, results for PBM are not as good as with HB. Also, results can vary accordingly to the layout of the Net. As we expected, moreover, the random search was particularly time-consuming.
The reported results are the best runs, and we reported the running time for the objective function too. The random search would have been extremely long without our max_iter stopping criterion, even though the use of `'CPLEX'` solver makes it faster than other solvers that we tried (e.g. `'HiGHS'`).
In our opinion, PBM is not the ideal choice for Neural Networks, neither in terms of convergence nor speed. There is no guarantee for a good convergence (see Section 4.), and this is probably why libraries like Keras won't even implement such a possibility for NNs. Analyzing the runs, we conclude that PBM is slower and less effective than HB at finding the optimal solution since our version is clearly made for convex problems. Nevertheless, if the required precision is not so high, this model is suitable anyway.

Furthermore, PBM appears to be much more sensitive to parameters μ and m_1. The most important parameter in terms of time is *max_sub*, and tables shows how higher values could lead the algorithm to convergence even more slowly (in terms of computational time) without an empirical increase in performances in terms of relative gaps.

References

[1] T.-M.-T. Do and T. Artieres. Regularized bundle methods for convex and non-convex risks. *Journal of Machine Learning Research*, 13(3539-3583), 2012.

[2] M. Dìaz and B. Grimmer. Optimal convergence rates for the proximal bundle method. *SIAM Journal on Optimization*, 33(2):424–454, 2023.

[3] C. Kizilkale. Topics in optimization: Restarted moment based methods, underdetermined systems and trade networks. page 8, 2019.

[4] T. B. T. P. Peter Ochs, Yunjin Chen. Ipiano: Inertial proximal algorithm for non-convex optimization. 2014.

[5] B. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, page 1–17, 1964.