KU LEUVEN

# Differentiable non-linear optimization as a layer in neural network architectures

Marco Salmistraro

Thesis submitted for the degree of
Master of Science in Artificial
Intelligence, option Engineering and
Computer Science

**Supervisor:**
Prof. Yves Moreau

**Assessors:**
Prof. Tias Guns
Antoine Passemiers

**Assistant-supervisor:**
Antoine Passemiers

Academic year 2022 – 2023

# Preface

A sincere feeling of gratitude goes to everyone who supported me through this Advanced Master's degree.
First of all to Antoine Passemiers, who relentlessly helped me throughout the research and coding phases of this project: our weekly meetings have been crucial in gaining a solid understanding of the topics and autonomously expanding differentiable gradient estimation to arbitrary non-linear cases.
I also want to acknowledge the prompt support I received from Brandon Amos in better grasping the mathematical developments underpinning the original *OptNet* solution.
Finally, a huge thank you goes to my girlfriend Coralie, who helped me out with the illustrations on this thesis and was on my side throughout the whole journey.

*Marco Salmistraro*

# Contents

# Abstract

The inspiration for this thesis comes from the 2017 paper of Amos and Kolter, [1] where their *OptNet* approach is proposed to solve QP problems in the context of NN architectures.

The representational power of typical fully-connected layers is somewhat limited by their own structure: passing information through an optimization layer, on the other hand, allows for integrating constraints and variable dependencies that would otherwise be hard to represent, while concurrently enforcing a form of implicit regularization on the network's ouptut.

This thesis first focuses on a comprehensive literature review on the topics of NNs, AEs and optimization frameworks. The applicability of differentiable optimization is then expanded from QP to wider categories of problems, up to the general NLP case. Finally, a practical simulation is presented in the form of approximated NLP embedded into an AE model; the developed `PyTorch` script allows to expand the results to arbitrary parameterizations of non-linear constrained optimization problems. Results are presented, discussed and perspectives for future work laid out.

# List of Figures and Tables

## List of Figures

## List of Tables

# List of Abbreviations

| | |
|---|---|
| ADAM | ADAptive Moment (estimation) |
| AE | AutoEncoder |
| AI | Artificial Intelligence |
| BFS | Basic Feasible Solution |
| CP | Convex Programming |
| DCG | Dynamic Computational Graph |
| DCP | Disciplined Convex Programming |
| FFT | Fast Fourier Transform |
| GAN | Generative Adversarial Network |
| GD | Gradient Descent |
| KKT | Karush-Kuhn-Tucker (conditions) |
| IPM | Interior-Point Method |
| LP | Linear Programming |
| ML | Machine Learning |
| MLP | Multi-Layer Perceptron |
| MSE | Mean Squared Error |
| NLP | Non-Linear Programming |
| NN | Neural Network |
| PSD | Positive Semi-Definite (matrix) |
| QP | Quadratic Programming |
| RBF | Radial Basis Function |
| QCQP | Quadratically Constrained Quadratic Programming |
| SDP | Semi-Definite Programming |
| SGD | Stochastic Gradient Descent |
| SLSQP | Sequential Least SQuares Programming |

# Chapter 1

# Introduction

Solving optimization problems in a correct and efficient manner is a key interest in the field of AI. In fact, these are an essential tool for modeling complicated phenomena and support fine-grained decisions on a variety of practical fields.

Among many application scenarios, optimization has been most notably employed as a tool for inference in learning processes: for example, in computer vision [42] or in finance [6]. Focusing on purely mathematical or ML-related tasks, there exist a variety of use cases which can greatly benefit from the integration of optimization problems [18]. In fact, these can efficiently be employed in classification, clustering, but also to regulate and optimize the convergence of GANs [33]. Other authors have also proposed their use for improved reinforcement learning: policies are refined in terms of progressive selection of feasible actions, defined as solutions to a constrained problem [44]. Interestingly, this is not a completely novel field: looking back in time, the adoption of specific classes of optimization problems within NNs had already been tackled by some authors many years ago [31].

The overarching motivation for this research work originated from the will of validating the use of constrained optimization in the context of NN architectures, while concurrently leveraging the *OptNet* approach and expanding its applicability to more general classes of problems. Overall, the aim is to allow for more flexible end-to-end learning as opposed to requiring user intervention for model tuning.

The first chapters introduce the mathematical framework for NNs, AEs and optimization problems. The thesis then draws inspiration from the work found in [1] to discuss constrained optimization frameworks within network architectures for different classes of problems. The final chapter is dedicated to practical applications of these concepts: results are presented and leveraged to obtain key insights and suggestions for future research.

## 1.1 Chapters 2–3

These two chapters are dedicated to setting the stage for the concepts discussed throughout the whole thesis. First, NNs are introduced and motivated from a mathematical point of view. Next, their structure is expanded into more complex architectures, thereby obtaining AEs. Chapter refcha:2 formalizes the concept of an optimization problem and provides mathematical descriptions for foundational topics in the same area; moreover, some well-known search algorithms are presented.

## 1.2 Chapters 4–5

First, chapter 4 discusses the scope and methodology for the work conducted in [1], which initially inspired this research work. The originally proposed solution strictly focuses on a batch implementation of convex QP problems; as such, chapter 5 attempts to extend the applicability of the original problem formulation beyond simple QP instances. Different classes of problems are defined and methodologies for incorporating them into more complex networks exposed.

## 1.3 Chapter 6

Finally, chapter 6 presents a hands-on case of differentiable optimization embedded into a learnable model. The proposed simulations approximate the solution of a non-linear problem in the context of an AE for input data reconstruction. The methodology and results are exposed and evaluated in light of the ideas discussed in the previous chapters.

# Chapter 2

# Neural networks

This chapter introduces many central concepts in the context of NNs; starting from a broad overview of their structure and applicability, a canonical formulation is provided and the most common techniques for operating with them are exposed.

## 2.1 Anatomy of a neural network

An artificial NN, or more commonly a NN, is a computing architecture inspired by the biological neural structures found in animals. The basic element of a NN is a *node* or *neuron*, which transmits signals to other nearby units via synapses, referred to as *edges*. The messages that are passed on are real numbers, while the output of a node is typically a non-linear evaluation of the sum of its inputs, done by means of an *activation function*. The relative importance of different messages in the network is set by means of *weights*, which can increase or decrease the strength of the signal they are linked to. Moreover, neurons can feature a threshold level under which they are not active. In a typical network, nodes are organized into layers: these perform different types of transformations on the input signals, traversing the network in a feed-forward fashion.

### 2.1.1 The perceptron model

The first computational model for a biological neuron was proposed in 1943 by Warren McCulloch and Walter Pitts [40]. From a formal point of view, it can be seen as a non-weighted sum of $n$ inputs to which a $b$ prior is added; this term is also referred to as *bias*. The model can be written as:

$$\hat{y} = \operatorname{sgn}(v^T x + b) = \operatorname{sgn}(w^T z), \tag{2.1}$$

with $\{x_i, y_i\}_{i=1}^n$ the training set while $x \in \mathbb{R}^n$ and $\hat{y} \in \mathbb{R}$ the input and output vectors, respectively. Indeed, it is rather intuitive to notice how a single perceptron separates the input space in a linear fashion. It is precisely because of this property that only a certain class of logic operators can be represented with it: the `AND` and `OR` function as an example; the `XOR` operator, on the other hand, cannot be correctly

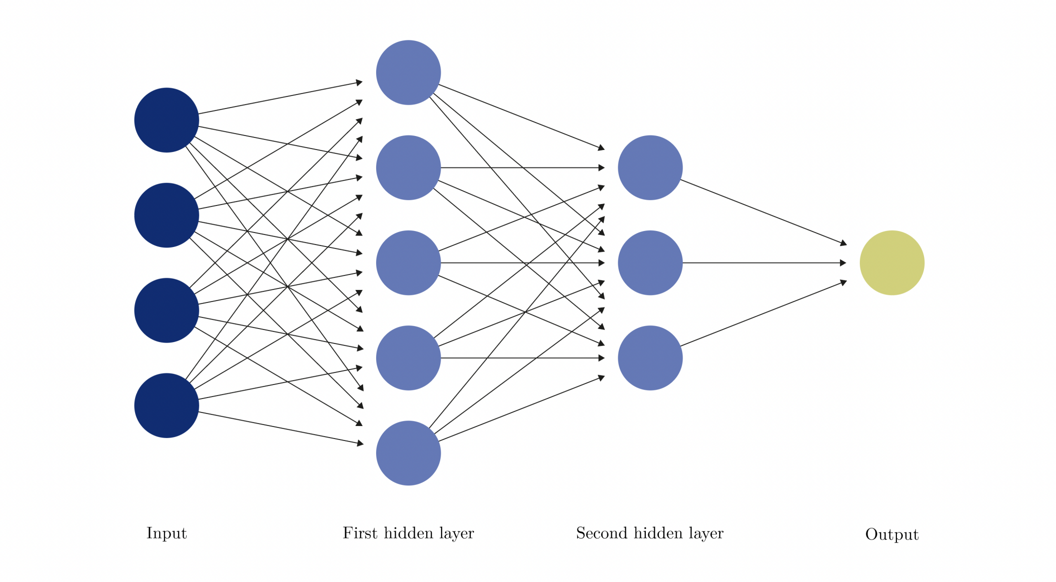

Figure 2.1: Schematic structure of a fully-connected neural network with two hidden layers. The 4-dimensional input is reduced to a one-dimensional output.

represented by a single unit. It is in light of these shortcomings that the model was further developed and *multi-layer perceptrons* were introduced. In fact, in the original statement of the perceptron, the adopted learning rule would only be applicable to single-node architectures. This called for reviewed techniques for weight learning, thus giving rise to the generalized *back-propagation* algorithm [50].

### 2.1.2 Formulation of a neural network

Let us consider a two-layer NN; let $x \in \mathbb{R}^n$ be the input vector, $y \in \mathbb{R}^m$ the output vector and $n_h$ the number of hidden neurons. Furthermore, let $W \in \mathbb{R}^{m \times n_h}$ and $V \in \mathbb{R}^{n_h \times n}$ be the interconnection weight matrices and finally $\beta \in \mathbb{R}^{n_h}$ the bias vector. The NN in question can be then formalized according to the following equation:

$$y_i = \sum_{r=1}^{n_h} w_{ir} \sigma(\sum_{j=1}^{n} v_{rj} x_j + \beta_r) \text{ for } i = 1, ..., m, \tag{2.2}$$

where with $\sigma$ we denote the activation function. Arbitrary extensions of the model in 2.2 can also be operated: for instance, by adding one more hidden layer it is possible to formulate the MLP (this time in matrix-vector form, for simplicity) as:

$$y = W\sigma(V_2\sigma(V_1 x + \beta_1) + \beta_2), \tag{2.3}$$

where $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $V_1 \in \mathbb{R}^{n_{h_1} \times n}$, $V_2 \in \mathbb{R}^{n_{h_2} \times n_{h_1}}$, $W \in \mathbb{R}^{m \times n_{h_2}}$ and finally, for the bias vectors, $\beta_1 \in \mathbb{R}^{n_{h_1}}$ and $\beta_2 \in \mathbb{R}^{n_{h_2}}$, with $n_{h_1}$ and $n_{h_2}$ the number of hidden neurons in the first and the second layer, respectively. Importantly, it must be noted that this formulation only applies to MLPs: an extension to generically non-linear $f^{(i)}$ functions of the $x$ input vector is possible by chaining operators into a nested structure, where $i$ indicates the $i$-th layer's index:

$$y = \bigcirc_{i=1}^{n} f^{(i)}(x). \tag{2.4}$$

### 2.1.3 Supervised learning

NN models can rely on a huge variety of algorithms for weight inference. The most common ones belong to the class of *supervised* methods: in order to learn a certain representation of the input data the network needs to be fed samples that it can learn from. In an *unsupervised* setting instead, the model is let evolve on its own without any specific input from the user, apart from hyper-parameter setting.

It is then practical to consider the working mechanism of a network architecture as a two-stage process: in a first moment, a set of samples $\{x_i, y_i\}_{i=1}^{n}$ is fed to the model. This is also the time for parameter tuning to take place via the back-propagation process (2.4): inputs are transferred through the network in a feed-forward fashion, until reaching the output layer. Then, by comparison with expected values, interconnection weights are adjusted as to minimize a specified loss function. This process takes place across an arbitrary number of iterations: data is generally split into patterns which are presented to the model in a sequential fashion, allowing for incremental weight adaptation.

A distinction can be traced between the *online* and *offline* learning paradigms: in the first case, weight learning takes place along with data generation; as such, the network is trained on new incoming data samples and constantly updated. The offline alternative expects that all patterns be contextually presented instead. As soon as all the input data has been fed to the model, an *epoch* is said to be complete; these are generally repeated hundreds or thousands of times over the course of a training loop. Finally, new data is fed into the network as to evaluate its performance and compare its output against the expected results.

### 2.1.4 Overfitting

Particular attention needs to be paid not to let the model train for too long, as this would correspond to incurring in a case of *overfitting*, an undesirable behavior where the architecture performs particularly well on the training set but shows poor performance on unseen data. Some possible approaches to the issue are:

- Tracking the evolution of the test error against training epochs. Identifying the point at which the model starts overfitting, then discarding all weight adjustments occurred after that point.

- Introducing regularization in the objective function. This ensures that weights be kept small during the learning process. Some of the most frequent types of regularization are $L1$ (Lasso), where the absolute value of each single $w_{ij}$

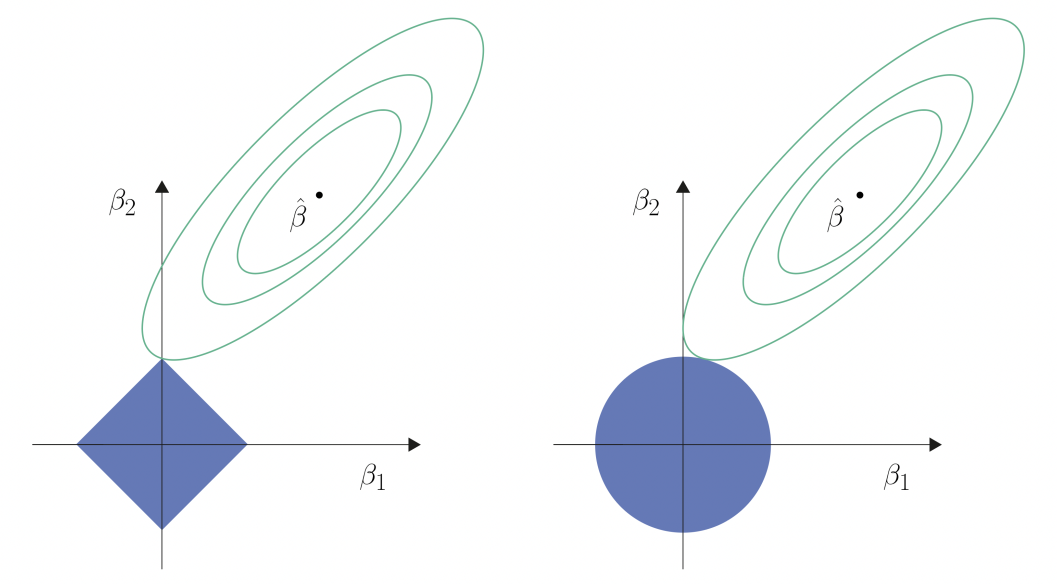

Figure 2.2: Relationship between loss function and regularization, seen as a 2D constrained optimization problem for the Lasso (left side) and Ridge (right side) cases. $\beta_1$ and $\beta_2$ represent the weight to be optimized while isocost lines for the $\hat{\beta}$ loss function are shown in green [26].

weight is taken to form $\Omega_1(W) = \sum_i \sum_j |w_{ij}|$, or $L2$ (Ridge) which considers the squared values for single weights, giving $\Omega_2(W) = \sum_i \sum_j w_{ij}^2$. In both notations $W$ represents the interconnection weight matrix.

- Applying the *drop-out* technique, whereby single neurons are deactivated according to a given probability.
- Addressing regularization at the early stages of a problem. In fact, this is the most efficient way to tackle overfitting issues; examples of this approach can be the definition of context-specific sparsity patterns but also the application of certain functions within a NN (as in the case of FFTs in speech processing applications). Moreover, this is exactly what constrained optimization layers aim at doing, since feasibility conditions are explicitly defined for training points.

Intuitively, regularization can be linked to the idea of deactivating certain neurons in the network: Lasso forces weights to decrease, eventually becoming $w_{ij} = 0$; Ridge operates similarly, by pushing weights to tend to 0 without never reaching it, at least from a theoretical point of view. What happens in practice is that weight can numerically be assimilated to 0, particularly in those cases where the corresponding feature is irrelevant or the regularization term is consistently high. Dropping out some neurons actually achieves the same result, ultimately simplifying the architecture and making it less prone to overfitting issues. This phenomenon can also be explained in terms of the *effective parameters* of a NN. Let $\lambda_j$ be the eigenvalues of the Hessian matrix for the energy function $E(W)$; the following equation can then be derived, where $\nu$ is the *regularization constant* [22]:

$$\tilde{w}_j = \frac{\lambda_j}{\lambda_j + \nu} w_j \quad \begin{array}{l} \text{if } \lambda_j \gg \nu \text{ then } |\tilde{w}_j| \simeq |w_j| \\ \text{if } \lambda_j \ll \nu \text{ then } |\tilde{w}_j| \ll |w_j|\,, \end{array} \tag{2.5}$$

where $\tilde{w}_j$ indicates the re-scaled version of the $j$-th weight according to the regularization constant $\nu$ and corresponding eigenvalue $\lambda_j$. As it turns out, higher values for $\nu$ correspond to a lower amount of effective parameters.

## 2.2 About representational power

The history of NNs dates back to the early 20th century, with the formulation of Hilbert's 23 problems. [24]. Most notably, the 13th problem is a conjecture regarding the impossibility of representing certain analytical functions in three variables as a finite superposition of continuous two-variable functions only. The first confutation came in 1957, with the formulation of Kolmogorov's theorem [34].

**Kolmogorov theorem (1957)** Any continuous function $f(x_1, ..., x_n)$ defined on the $[0,1]^n$ domain with $n \geq 2$ can be represented as

$$f(x) = \sum_{j=1}^{2n+1} \chi_j \left( \sum_{i=1}^{n} \phi_{ij}(x_i) \right), \tag{2.6}$$

where $\chi_j$ are continuous functions and $\phi_{ij}$ are both continuous and monotone.

This result was first refined by David Sprecher in 1965 [55] and later by the advent of the Hecht-Nielsen theorem [23], whose mathematical proof relied on Sprecher's theorem in order to trace a connection with MLPs.

**Hecht-Nielsen theorem (1987)** Any continuous function $[0,1]^n \to \mathbb{R}^m$ can be represented by a NN with two hidden layers.

A few years later, in 1989, a more powerful result was proposed by Hornik [25], who restricted the approximation requirement to just one layer.

**Hornik theorem (1987)** Given a certain metric, a certain activation function (not strictly continuous) and an $n$-dimensional input space, a one-layer NN can approximate any continuous function arbitrarily well.

In the following years these results have been generalized to other classes of activation functions, as for example RBFs [43]. All of these theorems grant the possibility of achieving non-linear modeling by means of a parameterization of non-linear functions and structures. Nonetheless, they do not prescribe any practical method for doing so: no explanation is given on how weights have to be set, nor on how many neurons are required by each layer. It is precisely in this context that the modeling power of NNs is most apparent.

## 2.3 Input dimensionality tolerance

NNs are not the only tool for universal approximation: similar results can be achieved by other mathematical objects like polynomial expansions, as granted by the Weierstrass theorem [57]. The preference for employing NNs is justified by their tolerance to high-dimensional spaces: under specific conditions, their approximation error becomes independent of the dimensionality of the input space [2].
Let us consider a perceptron where only one layer is present, namely featuring $n_h$ hidden neurons; the approximation error for a similar model is of order $O\left({}^{1}/_{n_h}\right)$. On the other hand, a polynomial expansion in an $n$-dimensional space and featuring $n_p$ terms would yield an error of order $O\left({}^{1}/_{n_p^{n^*}}\right)$ with $n^* = {}^{2}/_{n}$. This is perfectly in line with the idea that as the input dimensionality grows, the number of parameters or interconnection weights grows less dramatically for a MLP than it does for other approximators.

## 2.4 The back-propagation algorithm

A major breakthrough in the development of NNs was the introduction of the *back-propagation* algorithm [50], a supervised learning method for adapting interconnection weights and optimize a set loss function. It is worth looking in detail at the formal representation of the algorithm, as this is a central topic in the context of this thesis.
Let us consider a NN with $L$ layers denoted by an index $l$ as well as $P$ input and corresponding output patterns, indicated by index $p$. Let $n_l$ be the number of neurons for the $l$-th layer. The following equations are then established:

$$\begin{aligned} x_{i,p}^{l+1} &= \sigma(\xi_{i,p}^{l+1}), \\ \xi_{i,p}^{l+1} &= \sum_{j=1}^{n_l} w_{ij}^{l+1} x_{j,p}^{l}, \end{aligned} \tag{2.7}$$

where upper indices specify a layer in the network and lower ones refer to single neurons within a layer for a single pattern. In this notation, $x_{i,p}^{l}$ is the $i$-th component output by layer $l$ for pattern $p$. As for interconnection weights, $w_{ij}$ is the $i$,$j$-th entry of the interconnection matrix between layers $l$ and $l+1$. Finally, the activation function is denoted as $\sigma$; in terms of modeling choices, it is also possible to define different functions for each single layer.
The overarching goal of a NN is to minimize an *energy function*: although different alternatives are possible, a common choice is to use the MSE on the training set. The objective function is then minimized across all training patterns, according to:

$$\begin{aligned} \min_W &= \frac{1}{P}\sum_{p=1}^{P} E_p(W), \\ E_p &= \frac{1}{2}\sum_{i=1}^{n_l}(x_{i,p}^l - \hat{x}_{i,p})^2. \end{aligned} \tag{2.8}$$

Before introducing the general formulation for the back-propagation algorithm, the so-called $\delta$ variables need to be introduced; these are expressed as:

$$\delta_{i,p}^l = \frac{\partial E_p}{\partial \xi_{i,p}^l}. \tag{2.9}$$

It is then possible to write down the generalized *delta rule* defining the inner workings of the back-propagation algorithm:

$$\begin{cases} \Delta w_{i,j}^l = \eta \delta_{i,p}^l x_{j,p}^{l-1} = -\eta \dfrac{\partial E_p}{\partial w_{i,j}^l} \\ \delta_{i,p}^L = (\hat{x}_{i,p} - x_{i,p}^L)\sigma'(\xi_{i,p}^L) \\ \delta_{i,p}^l = \displaystyle\sum_{r=1}^{N_{l+1}} \delta_{r,p}^{l+1} w_{ri}^{l+1} \sigma'(\xi_{i,p}^l) \end{cases}, \tag{2.10}$$

where $\eta$ is the learning rate. As pointed out in 2.1.3, the learning process can either take place according to an online or offline paradigm.

## 2.5 Beyond the delta rule

In practice, there exist several alternatives to the pattern exposed in 2.10. Indeed, modern ML models commonly employ more efficient gradient-based optimizers in order to find optimal search directions and ultimately identify local minimizing points for the network's energy function. The setting can be seen as an unconstrained non-linear optimization problem for the $f(W)$ cost function, with $W$ representing the interconnection weight matrix. In the following paragraphs, different gradient-based alternatives for weight learning are presented.

**Gradient descent** This is one of the simplest approaches for identification of a suitable search direction. It can be summed up as:

$$W_{k+1} = W_k - \alpha_k \nabla f(W_k), \tag{2.11}$$

where $W_k$ is the interconnection matrix at the $k$-th iteration and $f$ is defined and differentiable in a neighborhood of the point where it is evaluated. As the desired target is to minimize the cost function, the approach consists in proceeding in a direction that is opposite to its gradient. By noting the starting estimation of the local minimum by $W_0$, a monotone sequence $f(W_0) \geq f(W_1) \geq f(W_2) \geq ... \geq f(W_n)$ is defined, where the aim is to obtain an estimate $W_n$ that is hopefully as close as

possible to the local minimum point.
The *learning rate* hyper-parameter $\alpha_k$ defines the size of each single step, and has to be carefully tuned in order to ensure that the search be efficient. Too low a value will cause the search process to be unnecessarily slow, with an excessive number of updates; on the other side, setting it too high could yield sub-optimal results by causing drastic updates to the minimizing point estimation and ultimately result in divergent behaviours. It must be noted that $\alpha_k$ does not necessarily need to be redefined at each single $k$-th iteration; as such, the $k$ index can be dropped. The convergence rate of this algorithm is linear; the next paragraphs explore faster, refined alternatives.

**Newton method** This alternative to simple GD allows for quadratic convergence to local minimum points. Let us recall equation 2.11 and consider an initial point estimate $W_0$; it is then possible to apply a truncated Taylor expansion in its proximity, specifically in the form:

$$f(W) = f(W_0) + \nabla f(W_0) + \frac{1}{2}\Delta W^T \nabla^2 f(W_0)\Delta W, \tag{2.12}$$

where $\Delta W = W - W_0$ is the search step, $\nabla f(W_0)$ is the gradient of $f$ at point $W_0$ and finally $\nabla^2 f(W_0)$ is the Hessian of the function at the same location. The optimal step $\Delta W^*$ can be then obtained by noting that the function needs to be stationary at any minimum point:

$$\frac{\partial f(W)}{\partial \Delta W} = \nabla f(W) + \nabla^2 f(W)\Delta W = 0 \Rightarrow \Delta W^* = -\left[\nabla^2 f(W)\right]^{-1} \nabla f(W). \tag{2.13}$$

**Levenberg-Marquardt method** A further alternative to the methods presented so far stems from the application of a constraint on the $\Delta W$ step, in the form of $\|\Delta W\|_2 = 1$. By concurrently expanding $f$ into a Taylor series the following Lagrangian is obtained:

$$\begin{aligned}\mathcal{L}(\Delta W, \lambda) = f(W_0) + \nabla f(W)^T \Delta W \\ + \frac{1}{2}(\Delta W)^T \nabla^2 f(W)\Delta W + \frac{1}{2}\lambda(\Delta W^T \Delta W - 1).\end{aligned} \tag{2.14}$$

From the optimality conditions one can then write

$$\begin{aligned}\frac{\partial \mathcal{L}(\Delta W, \lambda)}{\partial \Delta W} &= \nabla f(W) + \nabla^2 f(W)\Delta W + \lambda \Delta W = 0 \\ &\Rightarrow \Delta W^* = -\left[\nabla^2 f(W) + \lambda I\right]^{-1} \nabla f(W),\end{aligned} \tag{2.15}$$

which constitutes a particularly interesting result. Indeed, the $\lambda$ parameter regulates how much the algorithm resembles the Newton method or, rather, GD. For $\lambda = 0$ one obtains the same equation as for the first approach; conversely, for $\lambda \gg 0$ the result in 2.15 approximates the equation for GD search.

**Stochastic gradient descent** The original idea for this method can be traced back to the 1950s, with the research work of Robbins and Monro [46]: in essence it consists of a stochastic approximation of GD, where gradient calculations only make use of a part of the available points instead of the whole available data set.
Let us consider a data set of size $d$, on which GD is used for minimizing a certain loss function. If the model has $p$ parameters and the search process is run through $k$ iterations, the total number of computations to be performed can be expressed as $d \times p \times k$. SGD offers an alternative by picking one single point at a time from the given data set, thereby suppressing the $d$ term from the expression. Solutions for contextually sampling multiple points at the same time are also possible, and normally referred to as *mini-batch* GD. These are aimed at offering the robustness of GD coupled with the increased efficiency of SGD.

**ADAM** The underlying concept for adaptive moment estimation is to use squared gradients for scaling of the learning rate while building on a moving average estimate of the gradient in spite of the gradient itself. Individual learning rates are computed for each single parameter and updated according to first and second-order moments of the gradient of the loss function.
A $n$-th order momentum for a random variable $X$ is defined as $m_n = E[X^n]$; at each mini-batch iteration, ADAM computes moments for the mean and uncentered variance by exploiting moving averages of the gradient. The equations are defined as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \end{aligned} \tag{2.16}$$

where $m_t$ and $v_t$ are moving averages, $g$ is the gradient of the loss function for the current iterate, and finally $\beta_1$ and $\beta_2$ are hyper-parameters. Since the values in 2.16 are estimates, the following holds:

$$\begin{aligned} E[m_t] &= E[g_t], \\ E[v_t] &= E[g_t^2], \end{aligned} \tag{2.17}$$

which corresponds to stating that the expected values of the predictors are equal to the expected value of the parameters to be predicted. In the case of the ADAM method, however, the method is biased towards the initial value for the moving averages. By unrolling the iterative process for estimating parameter $m$ the formula for expressing a generic approximator at iteration $t$ is obtained. Note that a similar approach can be taken with regards to $v$, but is not presented here for brevity. The $m_t$ moving average can then be written as:

$$m_t = (1-\beta_1) \sum_{i=0}^{t} \beta_1^{t-1} g_i, \tag{2.18}$$

which then allows for inspecting the difference between the expected value for the moment and its true value:

$$
\begin{aligned}
E[m_t] =&E\left[(1-\beta_1)\sum_{i=0}^{t}\beta_1^{t-1}g_i\right]\\
=&E[g_t](1-\beta_1)\sum_{i=0}^{t}\beta_1^{t-1}+\zeta\\
=&E[g_t](1-\beta_1^t)+\zeta.
\end{aligned}
\tag{2.19}
$$

The first line of 2.19 is obtained thanks to 2.18; then, $g_i$ is approximated with $g_t$, thereby introducing the $\zeta$ bias term and allowing for taking the gradient term out of the summation operator. As a last step, the formula for a finite geometric series is employed.

Two main observations can be derived from this result. First of all, the obtained estimator is biased; furthermore, this bias is mostly appreciated at the beginning of the training loop, as the value of $\beta_1^t$ quickly and asymptotically approaches zero. The next step consist in applying bias correction to the two moving average estimations to obtain the $\hat{m}_t$ and $\hat{v}_t$ terms:

$$
\begin{aligned}
\hat{m}_t &= \frac{m_t}{1-\beta_1^t},\\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t}.
\end{aligned}
\tag{2.20}
$$

Finally, it is possible to express the generic $w$ weight update for any of the given model's parameters:

$$
w_t = w_{t-1} - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon},
\tag{2.21}
$$

where the learning step $\eta$ can depend both on the parameter to be updated and the iteration step.

## 2.6 Autoencoders

An AE is a special class of NN, composed of two distinct sections: a *coder* and a *decoder*. In the first step, the model learns a stripped-down representation of the input data, while in the second one the aim is operate a reconstruction by learning a function mapping the encoded representation back to the original input. In most cases, a similar mechanism is employed for dimensionality reduction tasks: indeed, the two learned functions can be seen as to be performing a compression of the input data with the underlying aim of keeping the reconstruction error as low as possible, as further explained in the next section.

Let us take a generic AE as an example; let $x \in \mathbb{R}^n$ be the input data for which a dimensionality reduction needs to be learned. Moreover, define its coder as a mapping $c : \mathbb{R}^n \rightarrow \mathbb{R}^m$, with $m < n$, while the decoder as $d : \mathbb{R}^m \rightarrow \mathbb{R}^n$. The intermediate $m$-dimensional representation is then said to take place in the *latent space*. Finally,

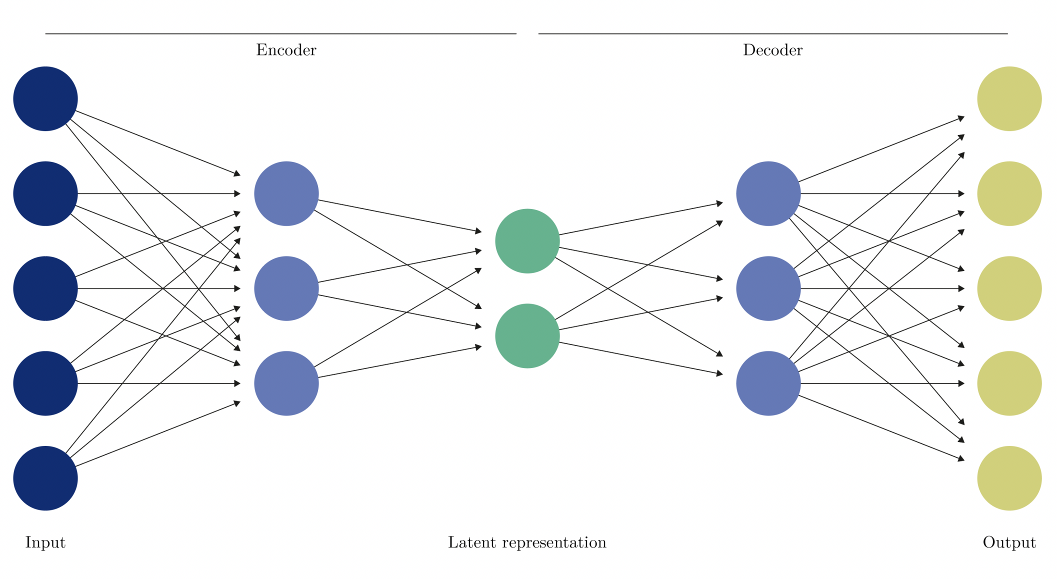

FIGURE 2.3: Schematic structure of a fully-connected autoencoder with one hidden layer on both the encoding and decoding networks. The lower-dimension latent space learns a compressed representation of the input.

it is convenient to equally denote with $E$ and $D$ the sets of possible encoders and decoders to be considered. This allows to write the dimensionality reduction task as

$$(e^*, d^*) = \underset{(e,d)\in E\times D}{\arg\min}\ \epsilon(x, d(e(x)))), \tag{2.22}$$

with $\epsilon(x, d(e(x))))$ the reconstruction error for input data $x$ and the encoding-decoding process involving mapping $d$ and $e$, respectively.

### 2.6.1 Training an autoencoder

The idea underpinning the training phase of an AE is to run an iterative process for optimizing the choice of the $e$ and $d$ mappings. At each iteration a new input vector is fed to the model, a reconstruction error computed and ultimately weights updated for both networks. The $\ell$ loss function to be minimized can then be formalized in the following way:

$$\ell(W_e, W_d) = \|x - \hat{x}\|^2 = \|x - d(\dot{x})\|^2 = \|x - d(e(x)))\|^2 \,, \tag{2.23}$$

where $W_e$ and $W_d$ indicate the interconnection weight matrices for the encoder end decoder networks, $\dot{x}$ is the learned representation in the latent space and finally $\hat{x}$ is the reconstructed input vector.

### 2.6.2 Unsupervised learning

Contrary to the typical supervised setting, which attempts at imparting a certain behaviour to the model by exposing it to a set of labeled data, *unsupervised learning* does not rely on any pre-existing knowledge of the input space. Instead, the model learns and organizes data without any explicit supervision, eventually learning clusters, patterns and representations that can find application in several domains: data exploration, anomaly detection or, as in the case of AEs, dimensionality reduction.

## 2.7 Conclusion

The present chapter has explored some basic concepts related to NNs; on top of that, alternatives for optimal direction search and weight adjustment have been defined and discussed. Finally, AE architectures have been introduced and techniques for their training explored. The coming chapters will focus on optimization problems, with the aim of employing them as a tool for designing end-to-end interaction in multi-layer networks.

# Chapter 3

# Optimization problems

This chapter introduces several key concepts in the field of mathematical optimization: these form the basis to later delve into case-specific techniques and discuss options for injecting different classes of problem formulations into layer-to-layer models. First, the typical elements of an optimization setting are defined. Then, core ideas such as convexity, KKT conditions and dual representations are discussed.

## 3.1 Anatomy of an optimization problem

In mathematics, the task of finding the best solution within a given set of feasible points is defined as *optimization*. At a macroscopic level, a distinction can be made between *discrete* optimization problems, where the variables to be optimized can take values from a finite set, and *continuous* ones, where variables can instead take values from a continuous domain. The presence of limits on the possible values for the variables determines whether the problem is *constrained* or not. More in detail, if such limits are multivariate, they are referred to as *constraints*; otherwise, it is customary to speak of *bounds*.
Finally, the notions of *feasibility* and *solvability* need to be touched upon. The first term refers to the function's domain subset defined by the applied equality and inequality constraints: in case this is empty, the problem is defined as *infeasible*. On the other hand, a problem is defined as solvable if an extreme point exists within the limits of the feasible set.

### 3.1.1 Continuous problems

The canonical form of a continuous optimization problem, as found in [5], is the following:

$$\begin{aligned} &\arg\min_x f(x) \\ \text{subject to } &g_i(x) \leq 0 \text{ for } i = 1, ..., m \\ &h_j(x) = 0 \text{ for } j = 1, ..., p, \end{aligned} \tag{3.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function to be minimized over the $x \in \mathbb{R}^n$ variable, $g_i(x)$ are the inequality constraints and $h_j(x)$ are the equality constraints; it also holds that $m, p \geq 0$. If both these two indices are zero, the problem can be defined as unconstrained.

### 3.1.2 Combinatorial problems

In case the domain of values to explore for the optimization process is finite or can be reduced to a discrete set, the search problem is referred to as a *combinatorial* one. In many practical applications, a complete, exhaustive exploration of the search space is not viable; therefore, algorithms have been devised to narrow down the set of candidate solutions to be evaluated. Many famous problems can be treated in terms of combinatorial optimization: the travelling salesman [47], but also the knapsack problem, whose first formulations date as back as 1896 [39]. From a formal standpoint, a combinatorial problem $P$ can be described as a tuple of four elements $P = (I, f, m, g)$ where:

- $I$ is a finite set containing instances $x$.
- For any given instance $x \in \mathbb{R}$, $f(x)$ represents the set of feasible solutions.
- For any given $x$ and $y \in f(x)$, $m(x, y)$ represents the *measure* of $y$.
- A *goal function* is defined as $g$ and either described as a minimization or maximization task.

As such, taking the case of a minimization setting, the problem can be formulated as

$$\arg\min_x m(x, y) = g\left\{m(x, y) \mid y \in f(x)\right\}. \tag{3.2}$$

Every combinatorial task can be transformed into its associated decision problem: this corresponds to asking whether a feasible solution exists for measure $m_0$. In the case of the travelling salesman, a possible decision problem would consists in determining whether a full trip can be completed in less than a given number of stops. This yields a binary solution domain: the answer can be either affirmative or negative.

Furthermore, optimization problems can sometimes be used to solve associated decision questions; the opposite can also be true, where decision problems can be extended to optimization frameworks. However, this approach is not guaranteed to give the best solutions in the feasible set.

**Differentiation of combinatorial solvers** Apart from the application cases that have already been mentioned, recent research has shown how the modeling power of combinatorial programming can be leveraged for state-of-the-art tasks. In [49] combinatorial solvers have been successfully employed in an end-to-end NN

architecture for deep graph matching and key-point correspondence in computer vision. One technically demanding aim of the research was to integrate differentiable gradient-based optimization in the architecture. Indeed, the mapping for the employed solvers matched continuous input data to a discrete output, defined as an indicator function, via piece-wise constant functions; this setting ultimately results in null gradients on most of the input space. The challenge was tackled thanks to some recent findings [45], proposing an *implicit interpolation* on the solver mapping, which stays otherwise untouched and is treated as a black-box. In practical terms, the forward pass is run on the intact network, whereas only a second call to the interpolated solver is necessary to retrieve gradient-related information.
Let $x$ be a continuous input and $Y$ a given discrete set; the solution proposed in [45] refers to optimization problems of the form

$$\begin{gathered} x \in \mathbb{R}^n \mapsto y(x) \in Y \subset \mathbb{R}^n \\ \text{such that } y(x) = \arg\min_{y \in Y} x \cdot y. \end{gathered} \tag{3.3}$$

Many traditional combinatorial problems can take this form, with graph matching being one of them. If $\ell$ is the loss function for the NN into which these mappings are incorporated, the suggested representation of the gradient for $w \mapsto \ell(y(x))$, which is a piece-wise function, can be written as

$$\frac{\mathrm{d}\ell(y(x))}{\mathrm{d}y} := \frac{y(x_\lambda) - y(x)}{\lambda}, \tag{3.4}$$

where term $x_\lambda$ is a modification of input $x$ depending on the gradient of $\ell(y(x))$. It is possible to show that this corresponds to an exact analytical representation of the gradient of a piece-wise mapping, where the hyper-parameter $\lambda$ regulates the interpolation range.

## 3.2 About convexity

Before diving deep into case-to-case applications of constrained optimization layers in the context of NNs, it is worth spending a few words on some basic concepts such as convexity, concavity and affinity. A function is defined as *convex* if a chord drawn from point $x$ to point $y$ lies entirely under the curve from $f(x)$ to $f(y)$. From an analytical point of view, this is equivalent to writing

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \tag{3.5}$$

for any $0 \leq t \leq 1$ and $x_1, x_2 \in X$. Conversely, a function is described as *concave* in case $-f(x)$ is convex. It is easy to see that linear problems are both convex and concave at the same time. Some other functions, on the other hand, are neither convex nor concave; this is the case of the sine function, for example.
A convex problem consists of both a convex objective function and convex constraints. In a similar setting, solving algorithms will either find a global optimum or prove that no feasible point exists: this intrinsic property allows them to be treated up to

very large numbers of dimensions. Non-convex problems, on the other side, present a totally different degree of difficulty, as the search for an optimum has to take place over a multi-faceted, complex landscape featuring multiple local minima. There exist several properties of convex functions that, in spite of being easy to demonstrate, are central to their analysis [48]. For a generic convex function it can be claimed that:

- All local minima are also global minima.
- The optimal set for the problem is a convex set: given any two points in the subset of dimension $d$, the line of dimension $d+1$ that joins them will equally fall within the set's boundaries.
- In case the function is strictly convex, the problem will result in at most one optimal point. The definition of a strictly convex (or conversely, concave) function comes from 3.5, where the inequality requirement states that the string connecting $x_1$ and $x_2$ needs to always be above the graph, except when joining the two points at its extremities.

Another important concept is that of affine functions: these can be represented as $f(x_1, ..., x_n) = A_1 x_1 + ... + A_n x_n + b$ where single $A_i$ coefficients are either scalars or matrices; at the same time, the constant term $b$ can either be a scalar or a column vector. This corresponds to mapping $n$-dimensional straight lines by performing a geometrical transformation, composed by a linear combination and a translation.

### 3.2.1 Disciplined convex programming

Convex programming is particularly interesting due to its several theoretical properties as well as the existence of diverse, reliable numerical algorithms for solution finding. On top of that, the range of fields where it finds application is wide and ever-growing. In terms of generalization properties, the development of efficient techniques for solving convex problems spans across both LP and QP, both subclasses of CP. Nonetheless, the level of expertise required to model CP instances is a considerable barrier to their widespread application: therefore, a formalized approach to design convex problems has been proposed under the name of *disciplined convex programming* [20], where general functions are expressed in terms of other functions whose curvature is known.
The major advantage of this framework is the possibility of following simple conventions for the constructions of DCP problems: in so doing, generality is preserved while providing a way of automating and enhancing instance formulation. More concretely, determining whether a certain problem is convex can be significantly hard; discriminating whether that same instance belongs to the class of DCP problems is instead straightforward. Finally, fully automated processes for conversion of DCP tasks to a solvable form are available: this allows to optimize their solution by taking advantage of their structure.

## 3.3 Primal and dual representation

An optimization problem can be approached from two different perspectives. Let 3.1 be the primal problem: if this is formulated as a minimization task, the dual is then necessarily a maximization instance. Since for any primal $p^*$ and corresponding $d^*$ dual solutions it holds that $p^* \geq d^*$ [5], any primal solution constitutes an upper bound to the dual; at the same time, any dual solution is a lower bound to its primal counterpart. This property of optimization problems is called *weak duality*; in case the primal is defined as a maximization task, it is easy to verify that $p^* \leq d^*$.
It is also possible to define a condition of *strong duality*, where the primal and dual solutions coincide: for convex problems this is always true under at least one constraint qualification condition, as listed in Table 3.1. However, it is generally not guaranteed that the primal and dual solutions coincide: therefore, the difference $p^* - d^*$ is defined and referred to as *duality gap*.

### 3.3.1 Duality in LP

Any LP problem can be conveniently converted into its dual form; by formalizing the canonical problem representation into matrix-vector form, it is possible to write the primal problem as

$$\begin{aligned} &\arg\min_x c^T x \\ \text{subject to } &Ax \leq b \\ &x \geq 0. \end{aligned} \tag{3.6}$$

The corresponding symmetric dual problem is then

$$\begin{aligned} &\arg\min_y b^T y \\ \text{subject to } &A^T y \geq c \\ &y \geq 0, \end{aligned} \tag{3.7}$$

whereas the asymmetric dual results from replacing the first inequality constraint in 3.7 with $A^T y = c$.
A first property of dual representations in LP problems is that the dual of a dual problem coincides with the original primal linear instance; this is however only valid for symmetric dual representations. On top of that, any point constituting a feasible solution in the primal corresponds to a bound on the optimal value for the dual. There might also be cases where the LP problem is not solvable or unbounded: in case the primal is unbounded, it follows from the weak duality principle that the dual problem is unfeasible. At the same time, the inverse applies: if the dual is unbounded, no solution can be found to the original primal instance. Finally, it can also be possible for both problems to be unfeasible, resulting in contextually empty primal and dual feasible sets.

### 3.3.2 Duality in NLP

Let us now focus on a general NLP case. Equation 3.1 can be then reconsidered by either taking $f(x)$ or any of the constraints to be non-linear. In case the domain of $f$ has a non-empty interior a Lagrangian function $\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$ can be defined, where the $\lambda$ and $\mu$ vectors are the *Lagrangian variables* or *multipliers* for the problem:

$$\mathcal{L}(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x). \tag{3.8}$$

Another function can be defined, namely the *Lagrange dual function*, as a mapping $l : \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$:

$$l(\lambda, \nu) = \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda, \nu) = \inf_{x \in \mathcal{D}} \left\{ f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x) \right\}. \tag{3.9}$$

Such expression has the interesting property of always being a concave function, even when 3.8 is convex; this is because it consists of a piece-wise infimum of affine functions. The dual representation also gives a lower bound on the optimal primal value $p^*$ for any problem: for any couple of $\lambda \geq 0$ and $\nu$ it is always true that $p^* \geq l(\lambda, \nu)$. More interestingly, in case the problem is convex and any constraint qualification condition as per Table 3.1 holds, it is guaranteed that

$$d^* = \max_{\lambda \geq 0, \nu} l(\lambda, \nu) = \inf f = p^*. \tag{3.10}$$

## 3.4 Solving an optimization problem in the dual

There are certain instances where solving the same optimization problem under a different formulation might bring about a number of advantages.

- First of all, it can be useful in terms of sensitivity analysis. Indeed, reformulating the problem in the primal representation by adding new constraints can easily make the original solution infeasible. On the other hand, the corresponding solution in the dual would still be feasible as the objective function would only change in its configuration or rather number of parameters.

- In the case of an iterative procedure where an initial feasible solution needs to be provided, it might be simpler to identify one according to the dual formulation.

- The dual might simply be easier to solve: a problem with many constraints and few variables can be easily converted into one with few constraints and many variables.

There exist different methods to operate with dual representations of optimization problems: the most simple case is that of an objective function whose feasibility is only determined by equality constraints.

### 3.4.1 Dealing with equality constraints

An optimization setting with no inequalities can be regarded as a special case of 3.1, where the $g_i(x)$ constraints are suppressed. A first step in the search of a minimizing point for $f(x)$ is to construct its Lagrangian, obtaining:

$$\mathcal{L}(x, \nu_j) = f(x) - \sum_{j=1}^{p} \nu_j h_j(x), \tag{3.11}$$

which allows for identifying the candidate optimal point. This is done by creating a system of equations yielding a saddle point:

$$\nabla\mathcal{L}(x, \nu_j) = 0 \Rightarrow \begin{cases} \dfrac{\partial\mathcal{L}(x, \nu_j)}{\partial x} = 0 \\ \dfrac{\partial\mathcal{L}(x, \nu_j)}{\partial \nu_j} = 0 \end{cases}. \tag{3.12}$$

At this stage, it is worth introducing a central theorem in the context of mathematical optimization, which gives more insight on the results obtained from 3.12.

**Lagrangian multiplier theorem** Take $f : \mathbb{R}^n \to \mathbb{R}$ to be the objective function of a constrained optimization problem, while $h : \mathbb{R}^n \to \mathbb{R}^c$ the constraining function; moreover, let both belong to $C^1$, indicating that they have continuous first-order derivatives. Denote by $x^*$ an optimal solution for the problem such that $\text{rank}(D(h(x^*))) = c < n$, with $D(h(x^*))$ the matrix of partial derivatives at point $x^*$, taking the form:

$$[D(h(x^*))]_{i,j} = \left[\frac{\partial h_i(x^*)}{\partial x_j}\right]. \tag{3.13}$$

Then for the constrained optimization problem

$$\begin{gathered} \arg\min_x f(x) \\ \text{subject to } h(x) = 0 \end{gathered} \tag{3.14}$$

there exists one unique Lagrange multiplier $\nu^* \in \mathbb{R}^c$ for which the following equality holds [10]:

$$Df(x^*) = [\nu^*]^T D(h(x^*)). \tag{3.15}$$

This is equivalent to stating that for any minimizing (or, conversely, maximizing) point for $f$ evaluated under equality constraints, in case constraint qualification holds it is possible to express the gradient of $f$ on that point as a linear combination of the gradients of the constraints, with single Lagrangian multipliers acting as coefficients.

This is effectively the same as observing that any direction that is perpendicular to all gradients of the constraints is also perpendicular to the gradient of the function itself, or that the directional derivative of the function is null in any feasible direction. Any solution to 3.12 is then guaranteed to be a locally optimal point according to the Lagrange method. This has the great advantage of not requiring any explicit parameterization in terms of the constraints. In order to take a step further and incorporate inequalities into the problem formulation, however, some other concepts need to be introduced.

### 3.4.2 Incorporating inequality constraints

After discussing a method for dealing with optimization problems bounded by equality-only constraints, it is time to expand the focus onto more general types of conditions for the optimization variables. It is then possible to bring back the $g_i(x)$ terms from 3.1 into the picture.

A first possibility requires inequality constraints to be in the form $g_i(x) \geq 0$: in this case, it is possible to simply plug them into the Lagrangian formulation of the problem and obtain a corresponding $\lambda_i$ multiplier for each single $i$ index, thus resulting in an expression analogous to 3.8. A second possibility is to introduce *slack variables*, thereby converting the inequality conditions from $g_i(x) \geq 0$ to

$$g_i(x) - s_i = 0, \tag{3.16}$$

where $s_i$ denotes the non-negative slack variable for the corresponding constraint. In doing so any inequality can be expressed as an equality and conveniently plugged into 3.8.

Once an optimum has been found, it is possible to classify equality constraints into two groups: active, for which the slack variable is $s_i = 0$ and a restriction on the objective function can be observed, and inactive ones, for which the corresponding Lagrangian multiplier is 0. These concepts are necessary to evaluate possible solutions to the saddle-point system represented in 3.12; of course, the requirement has to be extended to include inequality constraints as in $\nabla\mathcal{L}(x, \lambda_i, \nu_j) = 0$. As explained in the next section (3.21), the complementarity slackness condition states that for any inequality constraint evaluated in correspondence of any solution $x^*$ either the Langrangian multiplier is zero or the corresponding constraint is active. In practical terms, this constitutes a fundamental property whenever $\lambda_i$ or $\nu_j$ have to be retrieved from an iteratively optimized minimum, as discussed in 5.3.1.

## 3.5 About KKT conditions

Karush-Kuhn-Tucker conditions are a set of first-order derivative tests constituting a necessary requirement for a solution to any non-linear problem to be optimal; this holds under the premise that some regularity criteria be satisfied. Conversely, they are always sufficient to define a point as a local maximizer or minimizer. For a general optimization problem of the form

$$\begin{aligned} &\min_x f(x) \\ \text{subject to } &g_i(x) - b_i \geq 0 \text{ for } i = 1, ..., k \\ &g_i(x) - b_i = 0 \text{ for } i = k+1, ..., m, \end{aligned} \tag{3.17}$$

four different conditions can be written, with respect both to the primal and dual formulations. In case inequality constraints are absent, these are usually labeled as *Lagrangian conditions*, and the fourth criterion is dropped. In the notation employed here, both equality and inequality constraints are written as $g_i(x)$; furthermore, all Lagrangian multipliers are denoted with $\lambda_i$.

1. **Primal feasibility**: all constraints (equalities and inequalities) need to be satisfied. This means that for 3.17 it holds that

$$g_i(x^*) - b_i \text{ is feasible for } i = 1, ..., m. \tag{3.18}$$

2. **Stationarity**: no improvement on an optimal feasible point $x^*$ is possible. This can be written in a concise form according to

$$\nabla f(x^*) - \sum_{i=1}^{m} \lambda_i^* \nabla g(x^*) = 0. \tag{3.19}$$

3. **Dual Feasibility**: each single Lagrangian multiplier needs to be non-negative, resulting in a requirement for the constraints to be active. Therefore,

$$\lambda_i \geq 0. \tag{3.20}$$

4. **Complementarity slackness**: for each single constraint, the product of the Lagrangian multiplier $\lambda_i$ and the respective variable needs to be zero. This is equivalent to writing

$$\lambda_i^*(g_i(x^*) - b_i) = 0 \text{ for } i = 1, ..., m. \tag{3.21}$$

A set of properties can be derived with regards to KKT conditions and general optimization problems:

- They are both necessary and sufficient in convex problems.
- They are only necessary in the case of non-convex optimization, as in the case of a general NN training loop.

Let us consider a constrained minimization problem; let $f(x)$ be the target function and $x^*$ an optimal point. A natural question would be whether such point necessarily satisfies the KKT conditions; for an unconstrained problem, this is equivalent to asking under what conditions the point satisfies $\nabla f(x^*) = 0$. With regards to a constrained problem instead, a variety of increasingly complex conditions can be stated under which a minimizing point also satisfies the KKT conditions. Some of these are summed up in Table 3.1.

| Regularity conditions | | |
|---|---|---|
| Constraint | Acronym | Condition |
| Linearity constraint qualification | LCQ | If both $g_i$ and $h_i$ are affine functions, no further conditions are necessary. |
| Linear independence constraint qualification | LICQ | The gradients of the equality and inequality constraints, evaluated at $x^*$, are linearly independent. |
| Mangasarian-Fromovitz constraint qualification | MFCQ | The LICQ requirement is satisfied; also, a vector $d \in \mathbb{R}^n$ can be identified such that for all active inequality constraints it holds that $\nabla g_i(x^*)^T d < 0$ and for all active equality constraints $\nabla h_j(x^*)^T d = 0$. |
| Constant rank constraint qualification | CRCQ | All subsets of the gradients for both equality and inequality constraints have constant rank in an arbitrary vicinity of $x^*$. |

Table 3.1: Regularity conditions for constrained optimization problems. Under these conditions, constrained minima are also guaranteed to satisfy the KKT conditions.

## 3.6 Some algorithms for solving optimization problems

There exist several well-tested algorithms for tackling optimization tasks. Different classes of problems call for different approaches: this section is dedicated to introducing some of the most common techniques.

### 3.6.1 Simplex algorithm

The *simplex* method was originally developed by George Dantzig in 1947, while working for the US Army. Initially, the author formulated the problem as a series of linear inequalities with no objective function in a strict sense. In so doing, finding the optimal solution required applying pragmatic ground rules in order to limit the set of feasible solutions. One of the turning points in Dantzig's research was the idea of expressing such rules in terms of a linear target function to be maximized [7].
The methodology can be applied to LP problems of the form presented in 5.1: from a geometric standpoint, the feasible region for the problem consists of a convex *polytope*, which may be possibly unbounded; the extreme points or vertices of the region are defined as *basic feasible solutions*. A central insight offered by the method is that if a maximum point exists within the feasible region, then this has to be in correspondence of (at least) one of the extreme points. Since such points are finite in number, computation is also bound to take place in a finite amount of steps; however, in practice it generally holds that the amount of extreme points is unmanageable for all but the simplest instances of LP problems [41].
A second central idea is that in case a maximum is not found on one of the extremes of the polytope, then there exists an edge containing that point along which the

value of the objective function strictly increases in the direction moving away from it. As a consequence, two cases are defined: if the edge is finite, it connects to an extreme point with a higher value for the objective function; if it is not, the objective is unbounded and the problem has no solution.

It is possible to draw an easy-to-understand analogy with the simplex approach, where one walks along the edges of the polytope towards greater and greater values of the objective function, until a maximum is reached, as exemplified in Figure 3.1; alternatively, an unbounded edge is encountered, making the problem unsolvable. The algorithm always executes in finite time, as the number of points to be visited is also finite; furthermore, as the search direction is always pointing towards increasing values for the objective function, it is reasonable to expect the search space to be somewhat limited at each iteration.

There are two main steps making up the algorithm: in the first part, an extreme point is searched for in order to start the iterative process; this is normally done by solving a modified version of the LP instance. Consequently, it might either turn out that a BFS (an extreme point) is found or that the problem is infeasible: the latter happens whenever the feasible region defined by the polytope is empty. In the second step, the solution is refined by starting the search from the newly identified point; once again, two alternative situations are possible: either a new BFS is found or the edge is deemed to be infinite, thereby making the problem unbounded.

It is interesting to notice how the algorithm's name derives from an idea that is not strictly employed in the search process but is key to one of its possible interpretations: the method can be visualized as to operate on simplicial cones, which are turned into proper simplices by the addition of a further constraint; these effectively correspond to the vertices of the polytope defined by the problem's contraints.

In terms of efficiency, the simplex method is a well-known improvement on previously existing approaches, such as *Fourier-Motzkin elimination* [16]; still, it has been shown how worst-case time performance is exponential according to the original method formulation. It is an open question whether polynomial time variations are possible, although transformations that yield sub-exponential performance are known [21].

### 3.6.2 Primal-dual interior point methods

This class of solvers is useful in treating linear and nonlinear convex problems alike. The history of such methods dates back to 1967 [11]; the concept was later refined in several iterations until Narendra Karmarkar proposed its version of the algorithm in 1984 [29], allowing for solutions that would otherwise be impossible to find with the simplex method. On top of that, his approach is also proven to run in polynomial time. The algorithm's name comes from the fact that the feasible region is traversed in order to yield an optimal solution, thereby surpassing the conceptual limits of previous approaches.

Another valuable aspect of the method is the possibility of equally treating convex problems: this is achieved by encoding the corresponding set by means of a *self-concordant barrier function*. More specifically, a self-concordant function is defined by $|f'''(x)| \leq 2f''(x)^{3/2}$. A barrier function instead consists of a continuous mapping

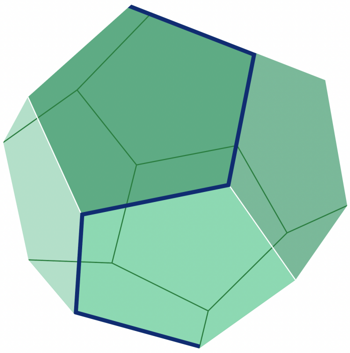

Figure 3.1: Three-dimensional representation of a polytope arising from a geometrical interpretation of the simplex algorithm. The blue lines show how the iterative search process moves from one extreme point to the other by following edges.

whose value increases to infinity as the argument approaches the boundary of the feasible region; this type of formalism is normally employed to replace inequality constraints with a penalizing term in the objective function, thus making operations easier to handle.

Further work by mathematicians Yurii Nesterov and Arkadi Nemirovski has shown how it is possible to encode any convex set by means of a barrier function, while also doing so in polynomial complexity [58].

### 3.6.3 Heuristic algorithms

The interest in this kind of algorithms resides in the impossibility or impracticality of applying exact inference to certain problem instances. Exact inference approaches typically yield feasible solutions that optimize the objective function: in this sense, they are granted to provide optimal points for the problem at hand. Heuristic alternatives, instead, identify feasible points with no optimality guarantee.

There are two main considerations that might suggest the application of heuristics: first of all, the underlying complexity of certain optimization settings; secondly, the time complexity of certain exact methods. This, however, does not mean that heuristic methods should be employed for all arbitrarily hard problems, but rather opted for when several concurrent challenges are encountered, including considerable data set sizes along with time and memory limitations. On top of that, a central requirement for employing heuristics is that a sub-optimal, approximate solution be acceptably good, that is, an absolute maximizer (or minimizer) in the feasible region is not strictly needed.

For certain classes of optimization tasks heuristics can be devised by drawing ideas from the problem's context: the effectiveness of the method is then tightly linked to how much the approach abides by the principles underpinning that specific problem or phenomenon. In this case it is customary to speak of *specific heuristics*. In recent years, particular attention has been paid by mathematicians and researchers to so-called *general heuristics approaches* [54], which overcome the domain-specific limitations of other approximated search strategies, and generally tend to outperform them. De Giovanni [9] proposes one of many possible subdivisions of such methods.

- **Constructive heuristics**: The solution is obtained by singling out the best-fitting item within a given subset. The process starts with the empty set, to which candidate points are added iteratively according to a specific optimality criterion. A characteristic feature of these approaches is that the selection made at a certain step only impacts future choices: as such, no backtracking is employed. Greedy-type algorithms, which operate according to local optimality considerations, belong to this class.

- **Meta-heuristics**: These algorithms are devised by abstracting away from specific problem formulations: they provide algorithmic schemes by defining their own components and interactions. These have to be later adapted to practical cases and applications by specializing each single component. Examples of approaches following this pattern are: *Simulated annealing* [32], *Ant colony optimization* [13] and *Particle swarm optimization* [30].

- **Approximation algorithms**: These are a special class of heuristics which operate under a set guarantee. Their potential resides in approximating the optimal solution up to a mathematically proven threshold. This, however, can end up being considerably large, thus posing an applicability limit.

- **Hyper-heuristics**: These methods stand at the forefront of AI and operations research, and aim at building models that can potentially generate and optimize algorithms themselves.

## 3.7 Conclusion

This chapter has formalized the concepts of optimization problems, convexity and primal-dual representations, among others. 3.6 sums up some popular approaches for specific cases of mathematical optimization. These ideas will be recalled and deepened in the rest of this thesis, as they are necessary for understanding the motivation and scope of the research. The next chapter builds on the first two by introducing and exploring the idea of differentiable optimization problems in the context of NNs.

# Chapter 4

# Constrained optimization in neural networks

This chapter discusses the incorporation of differentiable optimization problems into wider learnable architectures like NNs. It draws inspiration from the work of Amos and Kolter [1], which focuses on constrained minimization layers of the form:

$$
\begin{aligned}
\arg\min_{x} \; & \frac{1}{2}x^T Q(x_i)x + q(x_i)^T x \\
\text{subject to } & A(x_i)x = b(x_i) \\
& G(x_i)x \leq h(x_i),
\end{aligned}
\tag{4.1}
$$

where the optimization variable is defined as $x \in \mathbb{R}^n$ and a PSD matrix $Q \in \mathbb{R}^{n \times n} \succeq 0$ is employed. The problem parameters are $q \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $G \in \mathbb{R}^{p \times n}$ and finally $h \in \mathbb{R}^p$. These can depend on the previous $x_i$ layer as shown by the chosen notation; this is however a modeling choice and can be left out if one so desires.

When a similar framework is employed for modelling single layers, a solution to the problem is output as $x_{i+1}$: as a result, the forward pass in an *OptNet* architecture simply consists of solving the optimization problem stated in 4.1. Parameter adjustment is then operated during the backward pass and governed by the aim of minimizing the selected loss function.

## 4.1 Different approaches incorporating optimization

The work of Amos and Kolter does not represent a totally new research area, as many other authors before them have investigated the possibility of employing optimization frameworks to perform inference in complex learnable architectures. Some of the earliest works can be traced back to a couple of decades ago [31], [37]; other more recent results have also been published, notably in the fields of structured predictions [4], but also as a tool for achieving consistent denoising on image inputs [51]. [1] sketches out some common approaches that have been taken in scientific literature for training differentiable optimization parameters within wider neural models:

**Energy-based learning** These methods are underpinned by an arbitrarily defined energy function, shaped to assume low values in correspondence with the training data set. Although these approaches are well understood, they might also pose a number of challenges, notably in the form of instability issues [3]. On top of that, the training process always require data to be available, a further constraint that might not be easily addressed under certain operational conditions.

**Analytical approaches** In case an analytical solution can be found for the optimization task, gradients can be conveniently obtained and employed in layer-to-layer computations. This is however not possible in most practical cases as no exact solutions are known for a general constrained optimization problem.

**Loop unrolling** Another possibility is to approximate the minimizing operator by employing a first-order method and using gradient-based computations. An optimization setting is then enforced by selecting an appropriate search algorithm; this is then unrolled in order to obtain derivatives during the training process. A major drawback is the added complexity of the resulting network, which might clash with the available memory resources. Moreover, unrolling algorithms such as gradient descent are often convenient in the case of unconstrained settings; adding constraints for the definition of a feasible set can potentially require the adoption of a projection operator in the unrolling procedure, thereby making differentiation considerably harder.

**Operator differentiation** Some of the earliest works focusing on differentiating a minimizing operator stem from the area of *sensitivity analysis* [15]; other more recent ones focus on *bilevel optimization* tasks instead [36]. In the latter case, however, the authors describe a methodology for differentiating through equality constraints only and incorporate inequalities in the optimization framework by means of a barrier function. Differentiation on equality-only constraints is also proposed in [28], with a specific focus on convex target functions. In this case, the adoption of optimization frameworks is only relegated to the last layer of the variational network, whereas [1] puts forward a method for doing so at any given point within the architecture.

## 4.2 OptNet: differentiable optimization in QP

The method developed in [1] strictly focuses on solving QP problems characterized by a PSD matrix $Q$. In the authors' approach, forward traversal of the NN architecture consists of subsequent exact inferences on quadratic constrained optimization problems. This is conceptually rather simple and can be obtained by first defining parameters for the canonical formulation of the QP instance and then identifying an optimal feasible solution.

On the other side, the backward pass calls for a refined methodology, where the structure of the optimization problem is leveraged in order to infer gradients for weight updates. The authors differentiate the KKT conditions with respect to a given

solution to the quadratic problem, eventually obtaining an explicit formulation of the gradients of the loss function with regards to the QP input parameters. A major advantage of this procedure is that the backward pass over the optimization layer is calculated thanks to a specific factorization of the forward constrained problem. As such, no refactoring of the KKT matrix is necessary.
The mathematical details leading to explicit gradient expression in terms of the QP input parameters are voluntarily left out. Indeed, the simulations in chapter 6 only rely on the final results proposed in [1] and do not depend on any of the intermediate steps.

## 4.3 Implementing constrained optimization in `Pytorch`

Integrating optimization problems into NNs can be done by using any of the well-known Python libraries designed for layer-to-layer gradient operations: `TensorFlow` and `PyTorch` are just two of them. In the context of this project it is decided to employ `PyTorch` due to its ease of use as well as extensive library for tensor operations.

### 4.3.1 Automatic differentiation with `torch.autograd`

As introduced in chapter 2, the task underpinning any NN is to minimize a given loss function, measuring differences between predicted and expected outputs. During the forward pass, a first estimation of the output is computed, thus deriving a value for the loss function. Weights are then updated in a backward fashion according to a loss minimization paradigm and using any suitable algorithm, such as GD, Levenberg-Marquardt, quasi-Newton or stochastically-inspired ones. Crucial to this calculation is the learning rate hyperparameter, determining the size of the step taken by the search process. Steps are then repeated in an iterative way until a stop criterion is met.
What really drives the weight update is the calculation of gradients in the $\frac{\partial \ell}{\partial w_{ij}}$ form, with $w_{ij}$ the $i$,$j$-th component of the $W$ interconnection weight matrix; more precisely, a gradient is needed for each single parameter entry. As such, `PyTorch` creates a DCG for storing gradient calculations at every single iteration. In order to do so, custom functions can be specified, detailing both forward and backward operations; this allows for automatic tracking of the different tensor calculations and ultimately, for gradient estimation. Computations take place through the graph starting from the root (the output tensors) and are propagated to the leaves (the input tensors) by using the chain rule. A distinguishing feature of the `PyTorch` engine is that the DCG is created dynamically on the basis of the selected tensors that have been gradient-enabled and the operations or functions that are applied to them.

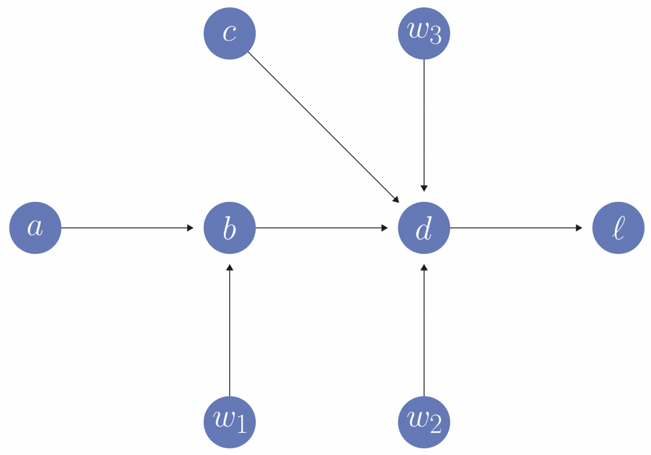

Figure 4.1: Schematic representation of the forward computation executed over a DCG. The graph is traversed in a feed-forward fashion, ultimately computing a value for the last $\ell$ node.

### 4.3.2 Dynamic computational graphs

In order to analyze the working mechanisms of DCGs, let us introduce a simple instance of NN with just four neurons, for which a set $\ell$ loss function is calculated, as shown Figure 4.1. The network can be entirely described by the following three equations:

$$\begin{aligned} b &= w_1 a, \\ d &= w_2 b + w_3 c, \\ \ell &= 100 - d; \end{aligned} \tag{4.2}$$

the optimization process takes then place over the domain space for weights $w_1$, $w_2$ and $w_3$. The application of the back-propagation algorithm requires that the gradients of $\ell$ with respect to these parameters be defined; let us then write down expressions for all three gradients:

$$\begin{aligned} \frac{\partial \ell}{\partial w_3} &= \frac{\partial \ell}{\partial d} \cdot \frac{\partial d}{\partial w_3}, \\ \frac{\partial \ell}{\partial w_2} &= \frac{\partial \ell}{\partial d} \cdot \frac{\partial d}{\partial w_2}, \\ \frac{\partial \ell}{\partial w_1} &= \frac{\partial \ell}{\partial d} \cdot \frac{\partial d}{\partial b} \cdot \frac{\partial b}{\partial w_1}. \end{aligned} \tag{4.3}$$

All three values have been computed through the chain rule; moreover, the individual terms on the right-hand side of the equalities can be obtained precisely

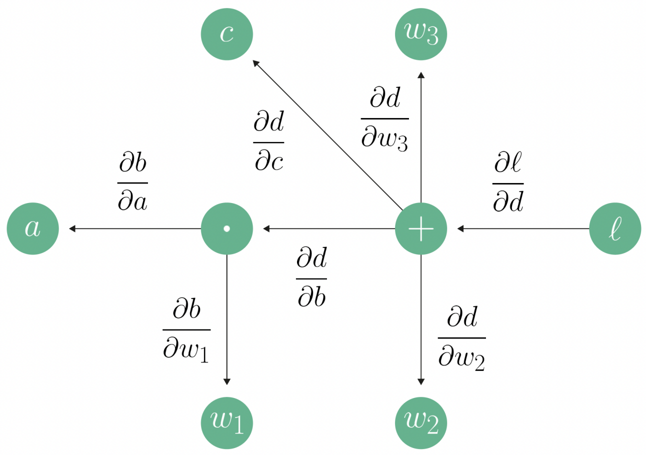
$c$
$w_3$
$\frac{\partial d}{\partial w_3}$
$\frac{\partial d}{\partial c}$
$\frac{\partial b}{\partial a}$
$\frac{\partial \ell}{\partial d}$
$a$
$\cdot$
$+$
$\ell$
$\frac{\partial d}{\partial b}$
$\frac{\partial b}{\partial w_1}$
$\frac{\partial d}{\partial w_2}$
$w_1$
$w_2$

FIGURE 4.2: Schematic representation of backward pass in a DCG. Gradients are propagated from the end node towards the beginning of the graph by leveraging the chain rule.

because the numerators can be expressed as explicit functions of the denominators. With reference to Figure 4.1, two nodes can be identified as *leaves*, namely $a$ and $c$, whereas $\ell$ is typically referred to as being the *root*: this is also the point from which backward gradient propagation starts. Each non-leaf node can be seen as a function taking some inputs and returning an output; in the case of $d$ the expression results in

$$d = f(w_2 b, w_3 c). \tag{4.4}$$

It is then easy to express the gradients of $f$ with respect to its inputs, as one obtains:

$$\nabla f = \left[ \frac{\partial f}{\partial w_2 b}, \frac{\partial f}{\partial w_3 c} \right]. \tag{4.5}$$

The same concept can be expanded across the whole graph, eventually obtaining the gradient expressions reported in Figure 4.2. At this stage the chain-rule can then be applied. Let us suppose that the gradient of $\ell$ with respect to node $a$ is needed; the procedure to be taken consists of three steps:

1. Trace all possible paths connecting $\ell$ to $a$.

2. For each single path, multiply all edges.

3. If more than one path is identified, sum the obtained products together.

Since Figure 4.1 only contains one path from $a$ to $\ell$, it is enough to operate a multiplication along all the edges connecting the two nodes:

$$\frac{\partial \ell}{\partial a} = \frac{\partial \ell}{\partial d} \cdot \frac{\partial d}{\partial b} \cdot \frac{\partial b}{\partial a}. \tag{4.6}$$

Implementing a similar computational model in `PyTorch` requires specifying methods both for the forward and backward passes. Let us proceed with the example of the $d$ node: in this case the forward method simply computes a weighted sum of the values stored in nodes $b$ and $c$, according to $w_2$ and $w_3$. On the other side, the backward pass takes term $\frac{\partial \ell}{\partial d}$ as an argument and outputs the gradients of $\ell$ with respect to $w_2 b$ and $w_3 c$; this is done by invoking the backward computation associated with the forward output vector ($d$ in this case), which is stored by `PyTorch` as an object attribute. Further down the line, the process continues in a recursive manner, with the next inputs to the backward method being $\frac{\partial \ell}{\partial w_2 b}$ and $\frac{\partial \ell}{\partial w_3 c}$.

### 4.3.3 Calculating the Jacobian-vector product

Whenever `PyTorch` needs to compute gradients for a loss function $l : \mathbb{R}^n \to \mathbb{R}^m$, a Jacobian matrix is never actually computed in the form of

$$J = \begin{bmatrix} \frac{\partial l_1}{\partial x_1} & \cdots & \frac{\partial l_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial l_m}{\partial x_1} & \cdots & \frac{\partial l_m}{\partial x_n} \end{bmatrix}, \tag{4.7}$$

but rather a different procedure is applied. Let us define $y = l(x) = (y_1, ..., y_m)$; this vector can then be used to compute the scalar loss $\ell$, ultimately obtaining the gradient tensor that is passed as an argument to the backward call:

$$v = \left[\frac{\partial \ell}{\partial y_1}, \ldots, \frac{\partial \ell}{\partial y_m}\right]^T. \tag{4.8}$$

In order to get the gradient of $\ell$ with regards to the model's parameters $W$, a matrix-vector product can be written as

$$J \cdot v = \begin{bmatrix} \frac{\partial y_1}{\partial w_1} & \cdots & \frac{\partial y_m}{\partial w_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial w_n} & \cdots & \frac{\partial y_m}{\partial w_n} \end{bmatrix} \begin{bmatrix} \frac{\partial \ell}{\partial y_1} \\ \vdots \\ \frac{\partial \ell}{\partial y_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial \ell}{\partial w_1} \\ \vdots \\ \frac{\partial \ell}{\partial w_n} \end{bmatrix}. \tag{4.9}$$

This method allows for easily employing external gradients via $v$, making application to non-scalar outputs promptly implementable.

## 4.4 Conclusion

This chapter has dealt with the concept of integrating constrained optimization settings within NN architectures. Moreover, some central insights regarding the working principles of the `PyTorch` engine have been offered. Chapter 5 will focus on expanding the applicability of similar approaches to different classes of optimization frameworks. Later, chapter 6 will demonstrate some practical scenarios for the application of differentiable optimization.

# Chapter 5

# Beyond quadratic problems

After understanding the rationale and applicability of *OptNet* layers in the context of purely convex quadratic problems, most of the research efforts discussed in this thesis have been focused on investigating the possibility of embedding optimization settings accounting for wider classes of problems. Several application cases are considered, with distinctions both based on the class of the objective function and the type of constraints. The simulations discussed in chapter 6 do not cover all possible classes of problems as presented here: rather, the non-linear extension aims at providing a method for tackling instances that are as general as possible. Let us then define the following classes of optimization problems:

- **Linear programming**: the objective function is linear in the optimization variable and so are the constraints.
- **Quadratic programming**: the objective function is both quadratic and convex, while constraints are linear.
- **Non-convex QP**: the objective function is non-convex quadratic; the constraints are linear, as the in the previous case.
- **Quadratically constrained quadratic programming**: both the objective function and the constraints are convex quadratic.
- **Non-convex QCQP**: the objective function is convex quadratic; constraints are also quadratic, but non-convex.
- **Non-linear programming**: either the objective function or at least one of the constraints are non-linear.

## 5.1 Linear programming

This implementation features constrained optimization problems in their simplest form. The canonical representation is:

$$\begin{aligned} \arg\min_x \quad & c^T x \\ \text{subject to } & Ax \leq b \\ & x \geq 0, \end{aligned} \tag{5.1}$$

where $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are parameter vectors and $A \in \mathbb{R}^{m \times n}$ is a given matrix. Of course, the same formulation can be applied to maximization-focused tasks by simply replacing the minimization operator.
The story of LP dates back to the 1930s, when authors like Kantorovich and Leontief first approached the topic with the aim of solving practical problems in the areas of logistics and economics [12]. Further down the line, with the advent of World War II, other possible fields of application became apparent, namely those of transportation and resource allocation. The renovated interest in LP greatly favored the development of new solving methods, culminating in 1947 with American mathematician Dantzig publishing a paper on his *simplex* method; this was further revisited and generalized as shown in [8].

### 5.1.1 Solving LP problems

LP problems can effectively be treated as a special case of QP, where the quadratic term is suppressed. Therefore, a viable option for embedding them into NNs is to leverage the original solution presented in the *OptNet* paper. The only caveat is to properly set $Q$ in order to eliminate the influence of the first term in the minimization expression. This can be conveniently done by choosing $Q = I$: the $I$ matrix is guaranteed to be PSD and thereby fulfill the convexity requirement.

## 5.2 Quadratic programming

A general quadratic programming problem can be defined as follows:

$$\begin{aligned} \arg\min_x \frac{1}{2} x^T Q x + c^T x \\ \text{subject to } Ax \preceq b, \end{aligned} \tag{5.2}$$

where the second line denotes component-wise inequalities. The typical requirement for the $Q$ matrix is for it to be symmetric; the approach taken in [1] however, considers it to be positive semidefinite: in other words, all of its eigenvalues need to be non-negative. As a result, the problem is transformed into a convex instance.

### 5.2.1 Solving QP problems

Finding a feasible solution for QP layers can be promptly tackled by using the *OptNet* approach. As stated above, attention must be paid to ensure that the quadratic term be PSD. Gradient calculation is then obtained thanks to the closed-form expressions provided in [1].

## 5.3 Non-convex QP

A natural extension of QP stems from suppressing the semidefinite nature of the $Q$ matrix, deriving a non-convex objective function. As for the constraints, these are still linear. As a reminder, a function is defined as convex if for any $0 \leq t \leq 1$ and $x_1, x_2 \in X$ the following is true:

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2). \tag{5.3}$$

### 5.3.1 Solving non-convex QP problems

Constrained optimization on non-convex quadratic target functions can be successfully embedded into NNs by tackling the usual forward and backward passes. First, layer-to-layer information is passed on by finding a solution to the optimization problem. This can be done, for example, with a non-linear solver algorithm like SLSQP: the solution is found across a sequence of successive quadratic problems under the assumption of constraint linearization [35]. The non-linear setting can be framed as follows:

$$\begin{aligned} &\arg\min_x \; f(x) \\ \text{subject to } &b(x) \geq 0 \\ &c(x) = 0. \end{aligned} \tag{5.4}$$

The formulation results in the Lagrangian $\mathcal{L}(x, \lambda, \nu) = f(x) - \lambda b(x) - \nu c(x)$ where $\lambda$ and $\nu$ are the Lagrange multipliers. At any iteration $k$ a solution $d_k$ is found as an optimal search direction, defined by the following QP problem:

$$\begin{aligned} &\arg\min_d \; f(x_k) + \nabla f(x_k)^T d + \frac{1}{2} d^T \nabla^2 \mathcal{L}(x_k, \lambda_k, \nu_k) d \\ \text{subject to } &b(x_k) + \nabla b(x_k)^T d \geq 0 \\ &c(x_k) + \nabla c(x_k)^T d = 0, \end{aligned} \tag{5.5}$$

where $x_k$ is the sub-optimal solution to build upon.

As for the backward pass, a different approach is needed, as the objective function is non-convex. Once again, the inspiration comes from [1]: in fact, a convex relaxation on the problem needs to be operated. This allows for approximating the gradients in the backward pass while maintaining exact inference of the optimal solution to the forward, non-convex problem. A possible technique is SDP (5.5.2); this is introduced in the context of more general problem instances (non-convex QCQP) but can be conveniently applied to the more restrictive QP case by making some simple assumptions.

Most of the available `Python` functions for implementing SLSQP typically return a $x^*$ minimized version of the input vector. On the other side, the corresponding Lagrangian multipliers are not promptly returned: a possible approach consists of solving a linear system involving the KKT conditions evaluated at $x^*$ in order to

retrieve $\lambda^*$ and $\nu^*$. This technique is also used in 6.1.
Naturally, it is also possible to employ other non-linear solvers for the resolution of the forward pass. Different algorithms can be applied, and their performances compared against each other. In case the chosen methodology converges to local minima it can be an option to employ heuristic search methods to obtain a first solution estimation and only then apply non-linear solvers for local solution refinement.

## 5.4 QCQP

This class of problems features convex quadratic objective functions along with equally convex quadratic constraints. *OptNet*-inspired integration into a NN can be simply achieved by employing any convex solver for the forward pass; gradient estimation during the backward pass, on the other side, requires rewriting the constraints after applying a linear approximation.

A QCQP problem can be expressed as:

$$\begin{aligned} &\arg\min_x x^T Qx + c^T x + b \\ &\text{subject to } x^T Q_i x + c_i^T x + b_i \leq 0 \text{ for } i = 1, .., m, \end{aligned} \tag{5.6}$$

where the problem variable is $x \in \mathbb{R}^n$ and the problem data consists of $m$ instances of $Q_i \in \mathbb{R}^{n \times n}$, $c_i \in \mathbb{R}^n$ and $b_i \in \mathbb{R}^n$, respectively. The problem can be conveniently described as convex in case all matrices $\{Q_i\}_{i=1}^m$ are PSD. In any other case, the instance does not satisfy convexity requirements. A discussion of the possible techniques to employ on non-convex frameworks can be found in 5.5.2.

### 5.4.1 Solving QCQP

As stated in the introduction to this section, the forward pass can be easily built by simply solving the underlying convex problem; this is conceptually rather simple and does not require any substantial re-working of the initial formulation. On the contrary, the backward pass calls for a linear approximation of the problem's constraints: this can be done by means of a truncated Taylor expansion.

### 5.4.2 Linear approximation of the constraints

Let $x_k$ be the optimal solution at the $k$-th iteration: since the point is feasible, a linear approximation of $g_i(x)$ gives the tangent to the constraint curve at point $x_k$:

$$g_i(x) \simeq g_i(x_k) + \nabla g_i(x_k)(x - x_k), \tag{5.7}$$

which, when translated into matrix-vector notation, can be re-written as

$$\begin{cases} A = \nabla g_i(x_k) \\ b = -\nabla g_i(x_k) x_k + g_i(x_k) \end{cases}. \tag{5.8}$$

## 5.5 Non-convex QCQP

This class of problems can be conceptually treated as an extension of the features found in 5.4. This is exemplified by formalizing the problem as in

$$\begin{aligned} &\arg\min_x x^T Q x + c^T x + b \\ &\text{subject to } x^T Q_i x + c_i^T x + b_i \leq 0 \text{ for } i = 1, .., m \\ &\text{where } \exists Q_i \in \{Q_i\}_{i=1}^m \text{ with } x^T Q_i x < 0. \end{aligned} \tag{5.9}$$

### 5.5.1 Solving non-convex QCQP problems

Approaching the integration of similar problems into NNs can be tackled by noting that, first of all, passing information in a feed-forward fashion calls for a non-linear solver. Then, weight adjustment via gradient back-propagation can be managed with a linear approximation of the quadratic constraints.

### 5.5.2 Convex relaxation of the objective function

In order to re-implement the backward pass according to the *OptNet* paradigm, non-convex problems require a relaxation of the objective function, whose $Q$ matrix would typically be indefinite. Many authors have tackled the topic and proposed practical approaches [17], [27].

**SDP** As reported by Eltved in [14], a common technique for dropping the non-convexity condition is to use SDP. The general representation of a QCQP problem is shown in 5.2; this can be conveniently converted into the so-called *Shor semidefinite programming relaxation* form [52], [53]:

$$\begin{aligned} &\arg\min_x \quad \frac{1}{2}\text{tr}(QX) + q^T x \\ &\text{subject to } \text{tr}(Q_i X) + {q_i}^T x + b_i \leq 0 \text{ for } i = 1, ..., m \\ &X \succeq x x^T. \end{aligned} \tag{5.10}$$

The advantage of 5.10 is that in case a minimizer $(x^*, X^*)$ is found for it, the condition $X^* = x^*(x^*)^T$ grants it the possibility of also minimizing 5.2. In so doing, the problem is reduced to a simpler class by just imposing one condition involving the optimization variable and the quadratic term.

**SDP Lagrangian duality** An alternative path to achieve SDP relaxation is by taking into account the Lagrangian dual form of 5.2, thus obtaining:

$$\mathcal{L}(x, \lambda) = x^T Q x + q^T x + \sum_{i=1}^m \lambda_i \left( x^T Q_i x + q_i^T x + b_i \right). \tag{5.11}$$

By following the steps shown in [14] the original non-convex QCQP problem can be reduced to a PSD constrained optimization setting that is the dual counterpart to 5.10. In case strong duality holds, it turns out that the same lower bound is given by both relaxation approaches.

## 5.6 Non-linear programming

Progressing onto wider generalizations for constrained optimization in NNs, it is possible to consider NLP cases. This class of problems is characterized by at least one of these two conditions: the objective function is non-linear or at least one of the constraints is non-linear.

### 5.6.1 Solving NLP problems

Integrating non-linear problems into a general framework for end-to-end layers is especially valuable, as it allows to maximize the degree of generalization at which requirements for feasible solutions are specified. A possible approach is to re-write the equations for the primal and dual representations reported in [1], by transitioning from a vector- and matrix-based notation to one referring to functions and corresponding gradients. This eventually yields a closed-form solution for backward gradients, which are expressed with reference to a specified parameterization.

A second alternative is to operate a quadratic approximation for the objective function and a linear one for constraints, as shown in 6.1.2, thereby obtaining a formulation that can be conveniently treated as a QP instance; a similar approach is taken for the NLP simulation instances presented in chapter 6.

## 5.7 Conclusion

After introducing several classes of optimization problems, this chapter has discussed techniques for their integration into wider NN architectures, detailing approaches for implementing forward and backward passes. The next chapter will focus on practical implementations of the concepts introduced so far, demonstrating how generally non-linear settings can be conveniently plugged into multi-layer optimization models.

# Chapter 6

# Simulations

This chapter focuses on hands-on implementations of the methods and techniques introduced over the course of this thesis. An artificial problem is dealt with, namely an instance of approximated NLP integrated into an AE architecture. First, a framework is provided in order to introduce the motivation, setting and methodologies for the trials. Next, results are presented and discussed. The corresponding code is available online and freely accessible [38].

## 6.1 Approximated non-linear programming

The aim of the simulations is to allow for generally non-linear optimization problems to be embedded into a wider deep-learning setting. It is relevant to observe how the potential range of applications for the architecture developed here might be somewhat limited: the interest lies in the integration of custom-defined NLP instances within layer-to-layer models. Most importantly, the obtained results demonstrate the possibility of specifying fine-grained requirements for the output layer, replacing the use of customary regularization terms in the loss function. In a wider sense, the use of constraints injects a form of implicit regularization into the model's structure.

### 6.1.1 Problem formulation

The problem is formulated as an AE instance, where the task at hand can be expressed in the following manner:

$$\min_{\theta} \mathbb{E}\left(\ell_{\theta}(x)\right) = \min_{\theta} \mathbb{E}(x - g(f_{\theta}(x)))^2. \tag{6.1}$$

A MSE measure is minimized: $x$ is the input vector, the encoder $f$ is represented by a NN parameterized by vector $\theta$ and $g$ consists of the selected constrained optimization problem. Variable $\dot{x}$ denotes the encoded representation obtained at the output of the NN, consisting of the learned values for the parameters of $g$. Finally, $\hat{x}$ is the reconstructed or decoded version of the $x$ input vector, for which it holds $\hat{x} = g(f_{\theta}(x))$. For the sake of the experiment, $g$ is taken to be a parameterized

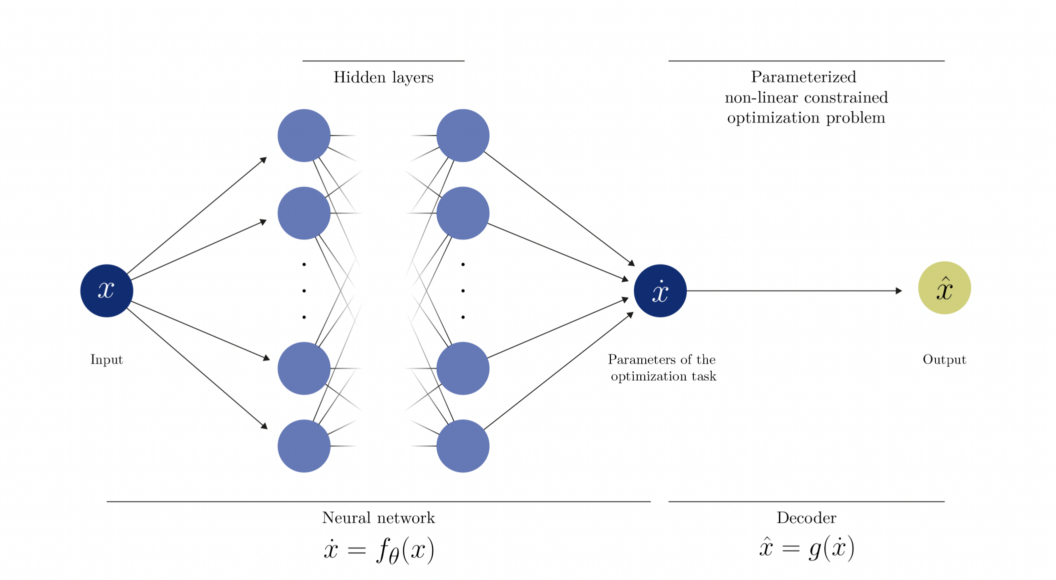

Figure 6.1: Structure of the model employed for embedding NLP into a wider AE architecture. The encoding stage passes input data $x$ through a NN. The decoder operates a reconstruction of the input vector by means of a non-linear constrained optimization task parameterized by the $\dot{x}$ vector learned in the latent space.

version of the *Rosenbrock function*, constrained by one cubic and one straight lines. The original problem formulation is:

$$\begin{aligned} f(x,y) &= (1-x)^2 + 100(y-x^2)^2 \\ \text{subject to } & (x-1)^3 - y + 1 \leq 0 \\ & x + y - 2 \leq 0. \end{aligned} \tag{6.2}$$

Different parameterizations are possible, both with regards to the objective function and the constraints. This has a subsequent effect on the dimensionality of the $\dot{x}$ vector, which stores the learned values for the chosen parameters. As an example, one could decide to plug variables $\alpha$ and $\beta$ into the objective function, or otherwise multiply the left-term of the first inequality constraint by an unknown term $\gamma$. Different options are evaluated in 6.1.3.

### 6.1.2 Methodology

When it comes to extending the applicability of the *OptNet* architecture to general NLP instances, one possibility is to re-work the mathematical developments presented in [1] by generalizing the equations contained in the paper to arbitrary non-linear functions, thereby obtaining a closed-form solution for exact inference of the parameters' gradients and weight updates. Instead, another option is presented here, where the objective function is approximated by a quadratic expression and

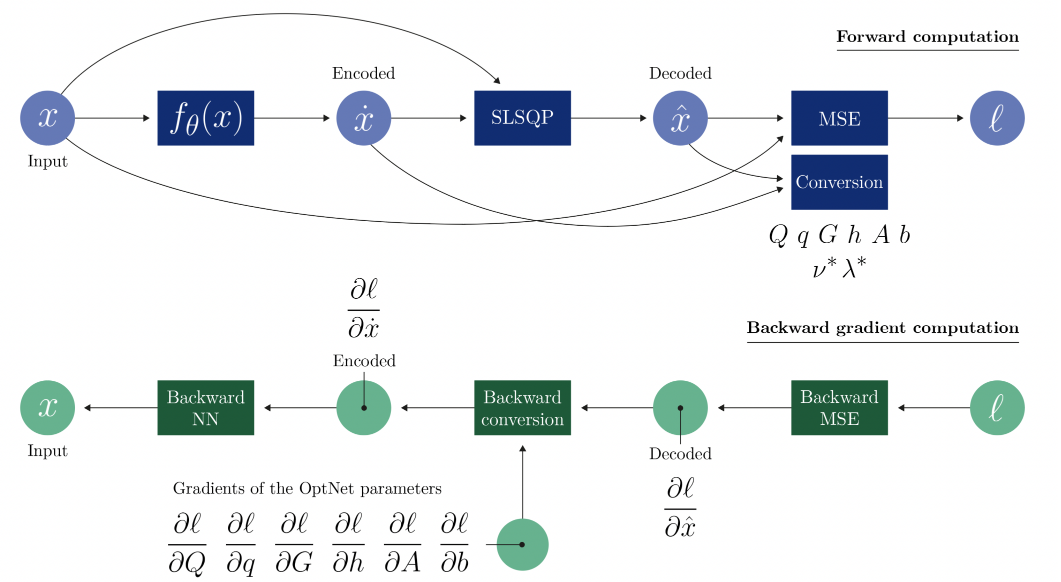
Forward computation
$x$
Input
$f_\theta(x)$
Encoded
$\dot{x}$
SLSQP
Decoded
$\hat{x}$
MSE
Conversion
$\ell$
$Q\ q\ G\ h\ A\ b$
$\nu^*\ \lambda^*$
$\frac{\partial \ell}{\partial \dot{x}}$
Backward gradient computation
$x$
Input
Backward NN
Encoded
Backward conversion
Decoded
Backward MSE
$\ell$
$\frac{\partial \ell}{\partial \hat{x}}$
Gradients of the OptNet parameters
$\frac{\partial \ell}{\partial Q}$ $\frac{\partial \ell}{\partial q}$ $\frac{\partial \ell}{\partial G}$ $\frac{\partial \ell}{\partial h}$ $\frac{\partial \ell}{\partial A}$ $\frac{\partial \ell}{\partial b}$

FIGURE 6.2: Schematic representation of the computations operated in order to embed *OptNet*-inspired approximations in the context of a NLP instance for the decoder stage of an AE.

constraints are re-written by means of a truncated Taylor expansion in the proximity of the considered point, as shown in 5.4.2. Since gradient estimation relies on the solution presented by [1], it is crucial to translate these approximations into the specific matrix-vector form that is employed in the paper.
In the following paragraphs, different aspects of the operational implementation of a solution in `PyTorch` are discussed. These pertain to topics of particular interest or relevance in this context.

**About the approach** Before diving deep into the mathematical assumptions and methodology adopted for the simulation, it is relevant to spend a few words on the underlying scheme employed for gradient calculation within a similar AE model. It must be noted that different approaches can be taken, particularly with regards to the quadratic and linear approximations; these will naturally yield different results. The approach chosen for this simulation is represented in Figure 6.2 and can be summarized as follows.

- The $x$ input vector is passed through the encoding stage of the model, which essentially consists of a multi-layer NN. This yields a corresponding $\dot{x}$ representation in the latent space.
- The elements of the $\dot{x}$ vector are the learned parameters for the NLP task in the decoding stage. As such, both $x$ and $\dot{x}$ are employed to compute a reconstructed version $\hat{x}$ of the original input via a non-linear solver. An alternative strategy

could be possibly taken here by employing a different starting point for the iterative SLSQP process.

- The objective function of the NLP instance is assumed to be quadratic in the proximity of $\hat{x}$; similarly, equality and inequality constraints are re-written by means of a linear approximation. In so doing, *OptNet*-like parameters are obtained ($Q$, $q$, $G$, $h$, $A$ and $b$).

- The obtained parameters are plugged into the KKT system evaluated at point $\hat{x}$, obtaining corresponding tensors for the optimal Lagrangian multipliers $\lambda^*$ and $\nu^*$. These are necessary to employ the closed-form *OptNet* expressions for backward gradients.

- Both $x$ and $\hat{x}$ are employed to compute the $\ell$ loss measure according to a set criterion; in this case, MSE is employed.

- Back-propagation is initiated from the $\ell$ node, employing the solution proposed in [1]. Since the original *OptNet* approach only computes gradients with respect to the approximation parameters, an edge must be explicitly defined in the DCG for computation of $\frac{\partial \ell}{\partial \hat{x}}$. This is done by writing methods for backward quadratic and linear approximations, which are respectively obtained by means of a vector-Jacobian product and a linear system.

**Making the differential matrix non-singular** The problem at hand features a generally singular differential matrix arising from the application of the *OptNet* framework. This is evident when following the approach taken in [1], where a square matrix is constructed to pre-multiply a differential vector and obtain one unique linear system for calculation of the parameters' gradients. It is then necessary to convert such a matrix into a suitable invertible form. This can be conveniently done by adding an arbitrarily small value to all of its diagonal entries. Notice that inverting a $n \times n$ square matrix $A$ requires that $\det(A) \neq 0$. By introducing a variable $t \in \mathbb{C}$ the determinant for the $A$ matrix can be rewritten as $\det(A - tI)$, which effectively corresponds to a polynomial expression with at most $n$ solutions. As $A$ is originally singular, $t = 0$ is a root for equation $\det(A - tI) = 0$. Therefore, it is possible to imagine a circle in the complex plane that is centered at the origin and contains no other solutions. For any $t \neq 0$ it is true that $\det(A - tI) \neq 0$, which is equivalent to stating that $A - tI$ is invertible. The same line of reasoning can be applied to real numbers by simply taking their absolute value to be arbitrarily small.

**Approximating the objective function and the constraints** Let us start by calculating the relationship between the linear Taylor expansion of a generic $g$ constraint and the $A$ matrix and $b$ vector of 4.1:

$$\begin{aligned} g(x) &\simeq g(x_0) + \nabla g(x_0)(x - x_0) \\ &= g(x_0) + \nabla g(x_0)x - \nabla g(x_0)x_0 \\ &= \nabla g(x_0)x - \nabla g(x_0)x_0 + g(x_0), \end{aligned} \tag{6.3}$$

which results in

$$\begin{cases} G = \nabla g(x_0) \\ h = -\nabla g(x_0)x_0 + g(x_0) \end{cases}. \tag{6.4}$$

A similar procedure can be followed to retrieve the parameters to be employed for approximation of a generic non-linear objective function.

$$\begin{aligned} f(x) &\simeq f(x_0) + \nabla f(x_0)(x - x_0) + \frac{1}{2}(x - x_0)^T \nabla^2 f(x_0)(x - x_0) \\ &= f(x_0) + \nabla f(x_0)(x - x_0) \\ &\quad + \frac{1}{2}\left[x^T \nabla^2 f(x_0)x - x^T \nabla^2 f(x_0)x_0 - {x_0}^T \nabla^2 f(x_0)x + {x_0}^T \nabla^2 f(x_0)x_0\right]. \end{aligned} \tag{6.5}$$

By noting that deriving constant terms produces null results, it is possible to discard some of the addends in 6.5. Indeed, the approximation is exclusively employed to compute gradients for the objective function relative to some of the problem's parameters. Therefore, the following simplification is obtained:

$$f(x) \simeq \nabla f(x_0)x + \frac{1}{2}\left(x^T \nabla^2 f(x_0)x - x^T \nabla^2 f(x_0)x_0 - {x_0}^T \nabla^2 f(x_0)x\right), \tag{6.6}$$

which translates into the following matrix-vector encoding required by the *OptNet* QP formulation:

$$\begin{cases} Q = -\frac{1}{2}\nabla^2 f(x_0) \\ q = \nabla f(x_0) - \frac{1}{2}\left([\nabla^2 f(x_0)x_0]^T + {x_0}^T \nabla^2 f(x_0)\right) \end{cases}. \tag{6.7}$$

**Calculating optimal Lagrangian multipliers in the forward pass** After obtaining the necessary parameters for writing the problem according to the *OptNet* formulation, it is equally necessary to retrieve the $\lambda^*$ and $\nu^*$ Lagrangian multipliers corresponding to the obtained $x^*$ optimal solution. This can be conveniently done by considering two of the KKT conditions for the problem, namely those for stationarity and complementarity slackness, which can be written in the form:

$$\begin{cases} Qx^* + q + A^T \nu^* + G^T \lambda^* = 0 \\ D(\lambda^*)(Gx^* - h) = 0 \end{cases}. \tag{6.8}$$

The system can then be considered as a linear problem whose unknowns are the Lagrangian multipliers. Let us then rearrange 6.8 into

$$\begin{cases} A^T \nu^* + G^T \lambda^* = -Qx^* - q \\ D(\lambda^*)(Gx^* - h) = 0 \end{cases}. \tag{6.9}$$

This can be further developed into another system of equations where $\nu^*$ and $\lambda^*$ are concatenated into vector $[\nu, \lambda]$, while matrices $A$ and $G$ are brought together into matrix $\mathscr{M}_{AG}$. In so doing one obtains:

$$\mathscr{M}_{AG} = \begin{bmatrix} A^T & G^T \end{bmatrix}, \tag{6.10}$$

which allows for expressing the first equation in 6.9 by virtue of the distributive property of matrix multiplication. In order to also account for the second equation in 6.9, it is necessary to focus on the $D(\cdot)$ operator, which maps a $n$-dimensional vector to a $n \times n$ square matrix where input elements are arranged on the main diagonal. A simple approach consists in noting that the following equality holds:

$$D(\lambda^*)(Gx^* - h) = D(Gx^* - h)\lambda^*. \tag{6.11}$$

The obtained diagonal matrix that pre-multiplies the optimal $\lambda^*$ value needs to be expanded with null entries to account for the full variable vector $[\nu, \lambda]$; let us denote the resulting matrix as $\mathscr{M}_D$. It is now possible to write the system in 6.9 into a formulation that can be treated by a LP solver:

$$\begin{cases} \mathscr{M}_{AG} \, [\nu, \lambda]^T = -Qx^* - q \\ \mathscr{M}_D \, [\nu, \lambda]^T = 0 \end{cases}. \tag{6.12}$$

Finally, the solution to this system yields the sought $\lambda^*$ and $\nu^*$ values, necessary for computing gradients in the backward pass of the model.

**Back-propagating the quadratic approximation** In order to back-propagate through the quadratic approximation of the NLP objective function around vector $\hat{x}$, one possibility is to compute a vector-Jacobian product: the Jacobian is defined on the quadratic approximation function taking as arguments the parameters obtained during the forward pass; the vector is instead constituted by the back-propagated gradients of $Q$ and $q$. Finally, the obtained tensor contains the gradients of the parameters of the objective function. In so doing, differentiability is maintained over the whole process. A different alternative is also practicable, where one exploits the approximation from 6.5 to build a quadratic equation. By considering a local approximation in the proximity of $\hat{x}$, a truncated Taylor expansion can be written, where $\dot{x}_{obj}$ denotes the $\dot{x}$ component containing the parameters for the objective function:

$$\begin{aligned} f\left(\frac{\partial \ell}{\partial \dot{x}_{obj}}\right) &\simeq f\left(\frac{\partial \ell}{\partial \hat{x}}\right) + \nabla f\left(\frac{\partial \ell}{\partial \dot{x}_{obj}} - \frac{\partial \ell}{\partial \hat{x}}\right) \\ &+ \frac{1}{2}\left(\frac{\partial \ell}{\partial \dot{x}_{obj}} - \frac{\partial \ell}{\partial \hat{x}}\right)^T \nabla^2 f\left(\frac{\partial \ell}{\partial \hat{x}}\right)\left(\frac{\partial \ell}{\partial \dot{x}_{obj}} - \frac{\partial \ell}{\partial \hat{x}}\right). \end{aligned} \tag{6.13}$$

By recalling 6.7 it is then straightforward to re-write this result as a quadratic equation with respect to $\frac{\partial \ell}{\partial \dot{x}_{obj}}$:

$$\left(\frac{\partial \ell}{\partial \dot{x}_{obj}}\right)^T Q \left(\frac{\partial \ell}{\partial \dot{x}_{obj}}\right) + q \left(\frac{\partial \ell}{\partial \dot{x}_{obj}}\right) - f \left(\frac{\partial \ell}{\partial \hat{x}}\right) \simeq 0. \tag{6.14}$$

A solution to a similar equation can be found by employing various approaches. In any case, specific requirements have to be analyzed case by case, as the $Q$ matrix is not guaranteed to be PSD; as such, one can either rely on a non-linear solver or alternatively operate a convex relaxation of 6.14. Most importantly, the dimensions of vectors $\dot{x}_{obj}$ and $\hat{x}$ need to coincide: an easily implementable trick is that of using null dummy values in correspondence of outstanding dimensions. Finally, obtained gradients for the objective function parameters can be propagated together with those resulting from backward linear approximations. However, in order to do so, a methodology for back-propagating constraints' parameters needs to be introduced.

**Back-propagating the linear approximations** Parameterizing the objective function of the constrained problem is not the only option for testing gradient propagation within the model. In fact, the code developed in [38] also allows the user to learn parameters for the inequality and, possibly, equality constraints.
First of all, the problem needs to be correctly formulated; as such, let us frame 6.2 in a slightly different way as seen before, by adding $\gamma$ and $\delta$ as parameters for the inequality constraints. Therefore, the $\dot{x}$ vector in the latent space is expanded from a 2-dimensional to a 4-dimensional array. At the same time, the implementation of the backward pass has to take into account information that was previously not propagated. By considering that a local linear approximation of the constraints yields parameters $G_i$, $h_i$ and $A_i$, $b_i$ for each single inequality and equality respectively, it is possible to construct a linear system of equations with the unknown variable being the gradient of the $\ell$ loss with respect to the encoded vector $\dot{x}_{cons}$. With reference to a NLP instance with $m$ inequality constraints, one obtains:

$$\begin{cases} \dfrac{\partial \ell}{\partial G_1}\dfrac{\partial \ell}{\partial \dot{x}_{cons}} + \dfrac{\partial \ell}{\partial h_1} = 0 \\ \dfrac{\partial \ell}{\partial G_2}\dfrac{\partial \ell}{\partial \dot{x}_{cons}} + \dfrac{\partial \ell}{\partial h_2} = 0 \\ \qquad\qquad\vdots \\ \dfrac{\partial \ell}{\partial G_m}\dfrac{\partial \ell}{\partial \dot{x}_{cons}} + \dfrac{\partial \ell}{\partial h_m} = 0 \end{cases} . \tag{6.15}$$

In the case where $m \geq 2$ the system is over-determined, calling for the application of a suitable algorithm; as an example, least-squares can be employed. In so doing, a complete path is traced from the partial derivatives of $\ell$ respective to the *OptNet* parameters to the partial derivative of the same loss function with respect to the interconnection matrix of the encoding NN. As explained in the previous paragraph, the dimensionality of $\dot{x}$ and $\hat{x}$ need to coincide in order to correctly leverage the linear constraint approximation.

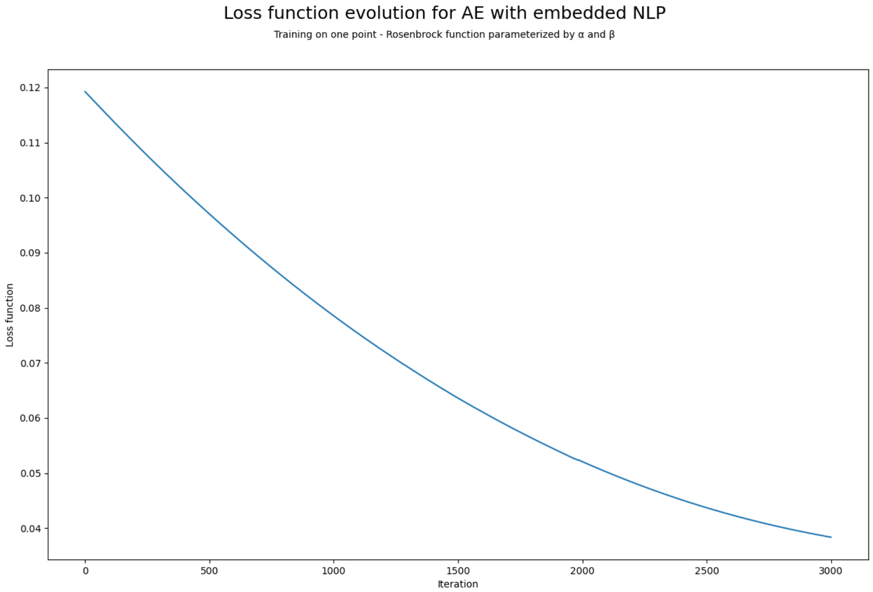

Figure 6.3: Convergence of the AE-embedded NLP instance, parameterized by 2 unknown variables $\alpha$ and $\beta$. Simulation on one single data point where only the objective function is parameterized; the progressive reduction in MSE suggests that gradients are correctly propagated and the model indeed learns the best-fitting parameters for the decoder.

### 6.1.3 Results

Hereafter, results from different runs of the model are presented and discussed. All training data for the following results was uniformly sampled from $[0,1] \times [0,1] \in \mathbb{R}^2$.

**First run** In order to verify whether the model converges, the choice is to run it for 3000 iterations on one single data point. The employed optimizer is ADAM with a learning rate `lr=1e-5`. The NLP formulation for the decoder is

$$\begin{gathered} f(x,y) = (\alpha - x)^2 + \beta(y - x^2)^2 \\ \text{subject to } (x-1)^3 - y + 1 \leq 0 \\ (x+y) - 2 \leq 0, \end{gathered} \tag{6.16}$$

where only the objective function is parameterized, namely by $\alpha$ and $\beta$. As visible in Figure 6.3, the model effectively converges to a suitable latent representation for the $[\alpha, \beta]$ vector.

**Second run** A second test is run on the AE model with the following parameterized version of the Rosenbrock function; in this case, a variable $\gamma$ is added to the picture to evaluate whether gradients are consistently computed for constraints as well:

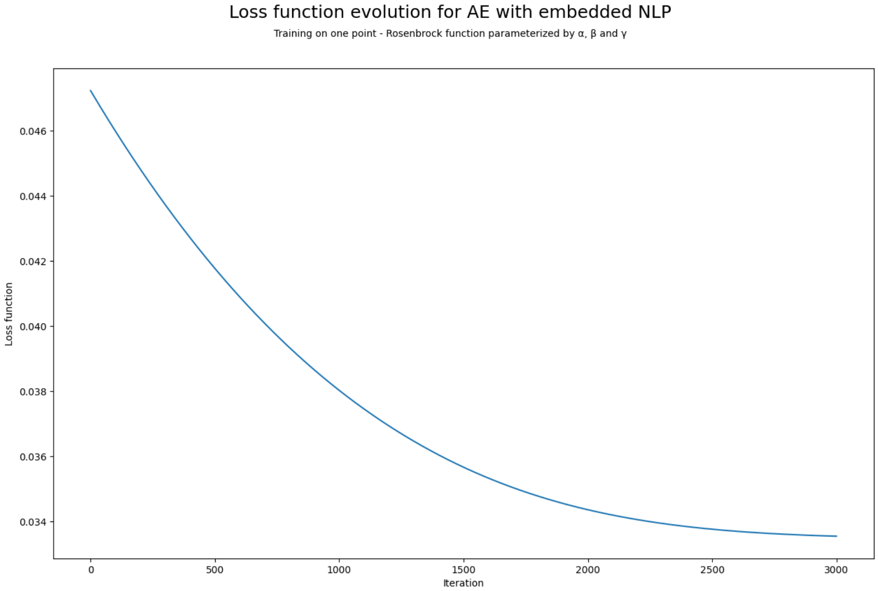

FIGURE 6.4: Convergence of the AE-embedded NLP instance, parameterized by 3 unknown variables $\alpha$, $\beta$ and $\gamma$. Simulation on one single data point where both the objective function and one constraint are parameterized; the exponential reduction in MSE suggests that gradients are correctly propagated and the model indeed learns the best-fitting parameters for the decoder.

$$\begin{aligned} f(x,y) &= (\alpha - x)^2 + \beta(y - x^2)^2 \\ \text{subject to } &(x-1)^3 - y + 1 \leq 0 \\ &\gamma(x+y) - 2 \leq 0. \end{aligned} \tag{6.17}$$

The hyper-parameters and training loop are set as in the first run. As visible in Figure 6.4, the model successfully learns the three parameters $\alpha$, $\beta$ and $\gamma$. Although the graph reported here shows a somewhat exponential decrease in MSE, a similar behaviour appears to be hardly consistent across different runs. A brief discussion on the variability of the obtained results is offered in 6.1.3.

**Third run** It is then decided to try and train the same three-parameter model on a bigger data set; as such, 50 points are uniformly sampled from the chosen $\mathbb{R}^2$ sub-domain and fed to the AE. It is reasonable to expect a somewhat slower convergence rate than previously seen, since the model needs to estimate optimal values for $\alpha$, $\beta$ and $\gamma$ by taking into account all of the training points. Indeed, this is exemplified by Figure 6.5, where the model is run for 6000 iterations with the same optimizer settings as before.

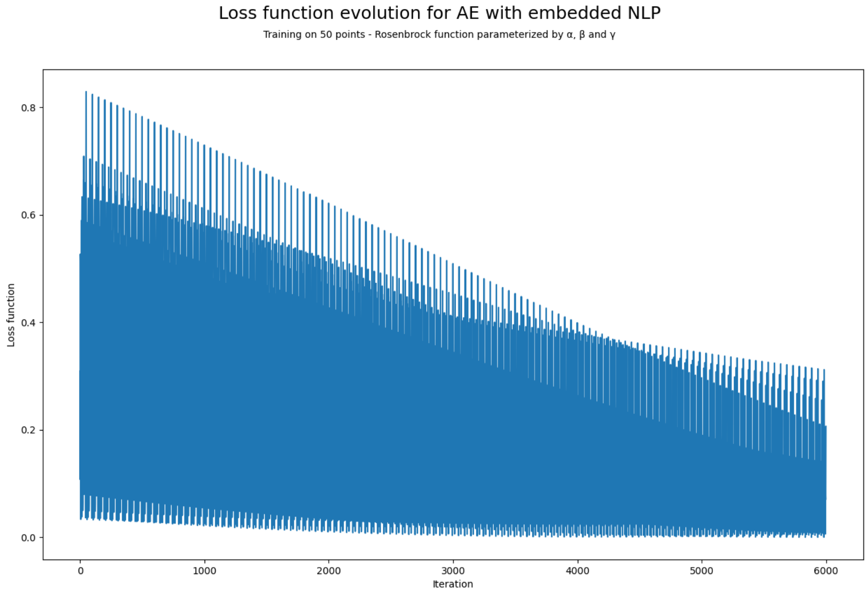

Figure 6.5: Convergence of the AE-embedded NLP instance, parameterized by 3 unknown variables $\alpha$, $\beta$ and $\gamma$. Simulation on 50 data points where both the objective function and one constraint are parameterized. The convergence rate is notably reduced compared to the first two runs; nonetheless, the model learns a better and better representation of the 3 parameters.

**Fourth run** A fourth test is performed by using a 50-point data set on a further modified version of the Rosenbrock function. This time, the rationale is to impose an even stronger, non-linear parameterization, with the aim to verify the model's behaviour. The following formulation is then chosen for the decoder:

$$\begin{aligned} f(x, y) &= (\alpha - x)^2 + \beta(\gamma y - x^2)^2 \\ \text{subject to } &(x-1)^{3\delta} - y + 1 \leq 0 \\ &(x + \epsilon y) - 2 \leq 0. \end{aligned} \tag{6.18}$$

Once again, the model shows an exponential convergence rate. Moreover, although the unit of measure for the chosen MSE criterion is somewhat arbitrary, as it simply consists of a squared difference between points in $\mathbb{R}^2$, it seems that a more complex parameterization of the decoder causes a higher training error: this can be seen by inspecting the MSE value over the first training steps, but also by noticing that convergence tends to stall at MSE $\geq 1$ after 6000 iterations.

**Stochastic elements** As it appeared from running the model multiple times, convergence is not always observed, neither in the one-point simulations nor in the case of extended data sets. Indeed, it turns out that keeping the data set unvaried across multiple runs can yield different results; in some cases the model does not even

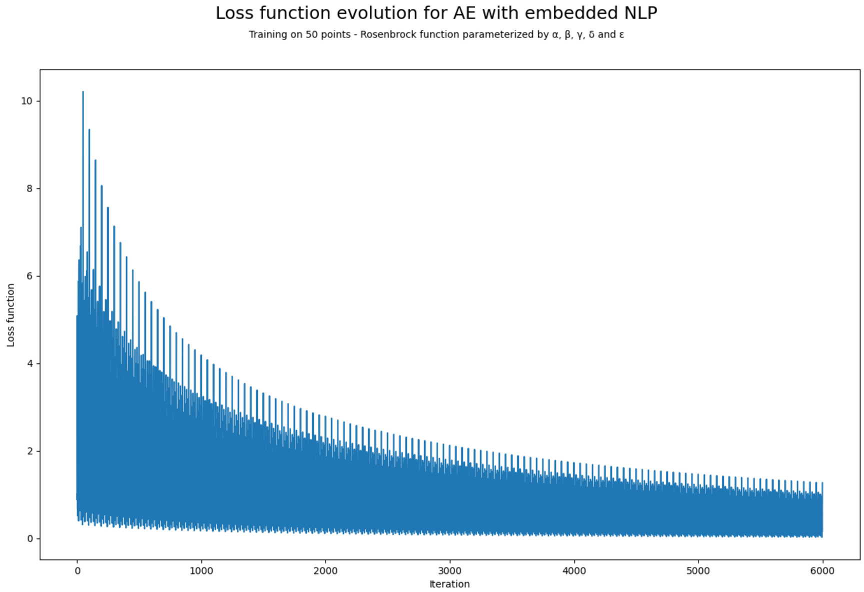

Figure 6.6: Convergence of the AE-embedded NLP instance, parameterized by 5 unknown variables $\alpha$, $\beta$, $\gamma$, $\delta$ and $\epsilon$. Simulation on 50 data points where both the objective function and the two constraints are parameterized. The model converges to an optimized representation of the 5 parameters.

convergence and considerable instability in terms of MSE evolution is observed. This is not a surprising result, as the training phase of a NN is typically a non-deterministic process [56]. However, it is worth looking into some elements of the model that could lead to this type of behaviour.

- Some of the initial experiments employed a randomization element for the choice of the initial solution estimate in the SLSQP solver. Although this could be a legitimate modeling option, it was later preferred to exploit input $x$ as a first approximated solution, following the scheme represented in Figure 6.2. This was done in the hope of reducing the observed stochastic behaviour; indeed, although at each run the data set is reconstructed by means of an uniform sampling procedure, the pseudo-random generator is seeded with a constant value.

- Weight initialization in the encoding NN takes place according to $w_{ij} \sim U(-\sqrt{k}, \sqrt{k})$, where $k = {}^{1}\!/_{\mathrm{dim}(x)}$. It is possible that the relative complexity of the AE model makes it somehow sensitive to this randomized sampling mechanism.

- It is reasonable to expect that the activation functions employed in the NN encoding stage could introduce some sort of chaotic behaviour during the training phase.

- Another item that could potentially introduce further randomness in the training phase is the SLSQP solver itself: cases are known where a non-deterministic behaviour arises [19].

## 6.2 Conclusion

The simulations covered in this chapter aim at proving the possibility of employing complex constrained optimization settings into learnable architectures. The first two results demonstrate how the respective model instances benefit from correct gradient computation and back-propagation, ultimately showing convergence. As explained in 6.1.3, the training loops only exploit one-point data sets, thus obtaining high-variance models which would be hardly scalable to unseen data. The results in Figure 6.3 and Figure 6.4 are however considered to be satisfactory: again, the goal is to demonstrate the feasibility of incorporating general non-linear settings into a NN, rather than creating a model that can be employed in a specific practical setting. Finally, the last two sample runs effectively show how more complex parameterizations can be learned, also providing an example of non-linear injection of variables in the constraints to the Rosenbrock's objective.
In spite of the satisfactory results obtained, stochastic behaviour was consistently observed across the different experiments, with arising gradient issues and diverging patterns, especially in the 5-parameter case. Still, the examples reported here prove the applicability of the developed framework for embedding of arbitrary NLP problems in end-to-end networks.

# Chapter 7

# Conclusions

As a conclusion to the present research, a first section is dedicated to discussing and summing up its most salient results. Then, a brief summary of possible extensions and focus areas for future work is offered.

## 7.1 About this thesis

The work presented so far efficiently demonstrates the possibility of injecting constrained optimization into wider models, such as NNs and AEs. A main motivation to taking up this thesis was the will to deep dive into the inner workings of `PyTorch` and its `torch.autograd` module: these have been thoroughly explored, as attested by the project's code base [38]. On top of that, the QP-only limitation of the original *OptNet* solution has been surpassed and expanded to general non-linear optimization instances and corresponding arbitrary parameterizations. The obtained results call for further exploration and study, as detailed in the next section.

## 7.2 Suggestions for future work

The experiments discussed in chapter 6 successfully prove the feasibility of employing constrained minimization settings as components for layer-to-layer architectures. Although the original results from [1] have been expanded to the general non-linear case, there remain several possible venues for further investigation.

- One of the main results of this thesis is the development of a framework for computing approximated backward gradients on general NLP instances. Although this class of problems already encompasses all other variants as expressed in chapter 5, deriving solutions for more specific cases could be a theoretically valuable exercise and shed some light on specific techniques to be adopted.

- As stated in 5.6.1, an alternative approach for the general NLP case could also be computed by finding a closed-form solution for a general non-linear task.

From a practical point of view this can be done by following the same steps as specified in [1]: the matrix-vector notation should then be replaced by generic, parameterized, non-linear functions. As a result, gradient computation in the backward pass would take place without involving any approximations.

- As it was the case in the original *OptNet* formulation, the developments presented in this thesis do not exploit any kind of sparse matrix operations. This could possibly constitute a valid option for reducing the computational load linked to optimization problem differentiation, especially in the case of considerably large data sets.

- Special attention could be paid to allowing backward gradient calculations to be performed according to a *parallel computing* paradigm. Although the examples presented in chapter 6 would hardly benefit from a similar implementation choice, the benefits would be considerable whenever handling bigger, high-dimensional data sets.

- The approximated non-linear solution exposed in 6.1 makes use of a non-linear solver for identification of the reconstructed vector that is output by the decoder. As in general non-convex problems a typical issue is the presence of multiple local minima, a viable option is to add a global optimizer to the local search. This can be performed by means of countless different heuristic approaches: a first global, possibly unfeasible solution could be identified and then refined by the local non-linear solver. The main hurdle to this type of implementation is the identification of a suitable way for expressing constraints, as heuristic optimizers are typically unconstrained.

- Different approaches for back-propagating the quadratic approximation of the objective function could be compared, both in terms of computational efficiency and convergence-related properties. As explained in 6.1.2, the dimensionality of the input and parameter tensors have to be tuned accordingly in order to correctly exploit the quadratic equation in 6.14.

- More research efforts could be dedicated to investigating the reasons for computational instability in models similar to the one proposed in chapter 6. On top of that, it would be interesting to understand whether there are possibilities for limiting the type of stochastic behaviour that was observed during the reported simulations.

- It would definitely be valuable to explore the computational efficiency of the non-linear approximation and gradient back-propagation methods taken in this thesis, while concurrently comparing it to other alternatives.

- Further programming efforts could be dedicated to developing a ready-to-use library for promptly and easily differentiating NLP instances in the context of neural models.

# Bibliography

[1] B. Amos and J. Z. Kolter. OptNet: differentiable optimization as a layer in neural networks. In *International conference on machine learning*, pages 136–145. PMLR, 2017.

[2] A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on information theory*, 39(3):930–945, 1993.

[3] D. Belanger and A. McCallum. Structured prediction energy networks. In *International conference on machine learning*, pages 983–992. PMLR, 2016.

[4] D. Belanger, B. Yang, and A. McCallum. End-to-end learning for structured prediction energy networks. In *International conference on machine learning*, pages 429–439. PMLR, 2017.

[5] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization.* Cambridge university press, 2004.

[6] G. Cornuejols and R. Tütüncü. *Optimization methods in finance*, volume 5. Cambridge university press, 2006.

[7] G. B. Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. 1990.

[8] G. B. Dantzig, A. Orden, P. Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific journal of mathematics*, 5(2):183–195, 1955.

[9] L. De Giovanni. Methods and models for combinatorial optimization: heuristics for combinatorial optimization. 2017.

[10] A. De la Fuente. *Mathematical methods and models for economists.* Cambridge university press, 2000.

[11] I. Dikin. Iterative solution of problems of linear and quadratic programming. In *Doklady akademii nauk*, volume 174, pages 747–748. Russian academy of sciences, 1967.

[12] R. Dorfman. The discovery of linear programming. *Annals of the history of computing*, 6(3):283–295, 1984.

[13] M. Dorigo. Optimization, learning and natural algorithms. *PhD Thesis, Politecnico di Milano*, 1992.

[14] A. Eltved. *Convex relaxation techniques for non-linear pptimization*. PhD thesis, PhD thesis, Technical university of Denmark, 2021.

[15] A. V. Fiacco and Y. Ishizuka. Sensitivity and stability analysis for non-linear programming. *Annals of operations research*, 27(1):215–235, 1990.

[16] J. Fourier. Histoire de l'académie, partie mathématique. *Mémoire de l'académie des sciences de l'institut de France*, 1824.

[17] T. Fujie and M. Kojima. Semidefinite programming relaxation for non-convex quadratic programs. *Journal of global optimization*, 10:367–380, 1997.

[18] C. Gambella, B. Ghaddar, and J. Naoum-Sawaya. Optimization problems for machine learning: a survey. *European journal of operational research*, 290(3):807–828, 2021.

[19] GitHub. The SLSQP solver gives stochastic behaviour. URL: https://github.com/scipy/scipy/issues/8677, last checked on 2023-08-13.

[20] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming. *Global optimization: from theory to implementation*, pages 155–210, 2006.

[21] T. D. Hansen and U. Zwick. An improved version of the random-facet pivoting rule for the simplex algorithm. In *Proceedings of the forty-seventh annual ACM symposium on theory of computing*, pages 209–218, 2015.

[22] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[23] R. Hecht-Nielsen. Kolmogorov's mapping neural network existence theorem. In *Proceedings of the international conference on neural networks*, volume 3, pages 11–14. IEEE press New York, NY, USA, 1987.

[24] D. Hilbert. Mathematische probleme. *Nachrichten von der koniglichen gesellschaft der wissenschaften zu gottingen*, 1900.

[25] K. Hornik, M. Stinchcombe, and H. White. Multi-layer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[26] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[27] J. Jancsary, S. Nowozin, and C. Rother. Learning convex QP relaxations for structured prediction. In *International conference on machine learning*, pages 915–923. PMLR, 2013.

[28] M. J. Johnson, D. K. Duvenaud, A. Wiltschko, R. P. Adams, and S. R. Datta. Composing graphical models with neural networks for structured representations and fast inference. *Advances in neural information processing systems*, 29, 2016.

[29] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on theory of computing*, pages 302–311, 1984.

[30] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[31] M. P. Kennedy and L. O. Chua. Neural networks for non-linear programming. *IEEE Transactions on Circuits and Systems*, 35(5):554–562, 1988.

[32] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[33] N. Kodali, J. Abernethy, J. Hays, and Z. Kira. On convergence and stability of GANs. *arXiv pre-print, arXiv:1705.07215*, 2017.

[34] A. N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In *Doklady akademii mauk*, volume 114, pages 953–956. Russian academy of sciences, 1957.

[35] D. Kraft. A software package for sequential quadratic programming. *Forschungsbericht- deutsche forschungs- und versuchsanstalt fur luft- und raumfahrt*, 1988.

[36] K. Kunisch and T. Pock. A bi-level optimization approach for parameter learning in variational models. *SIAM journal on imaging sciences*, 6(2):938–983, 2013.

[37] W. E. Lillo, M. H. Loh, S. Hui, and S. H. Zak. On solving constrained optimization problems with neural networks: a penalty method approach. *IEEE transactions on neural networks*, 4(6):931–940, 1993.

[38] Marco Salmistraro. Differentiable optimization repository. URL: https://github.com/marcosalmistraro/differentiable_optimization, last checked on 2023-08-13.

[39] G. B. Mathews. On the partition of numbers. *Proceedings of the London mathematical society*, 1(1):486–490, 1896.

[40] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[41] K. G. Murty. *Linear programming.* Springer, 1983.

[42] P. Ochs, A. Dosovitskiy, T. Brox, and T. Pock. On iteratively re-weighted algorithms for non-smooth non-convex optimization in computer vision. *SIAM journal on imaging sciences*, 8(1):331–372, 2015.

[43] J. Park and I. W. Sandberg. Universal approximation using radial-basis function networks. *Neural computation*, 3(2):246–257, 1991.

[44] T.-H. Pham, G. De Magistris, and R. Tachibana. OptLayer - practical constrained optimization for deep reinforcement learning in the real world. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 6236–6243. IEEE, 2018.

[45] M. V. Pogančić, A. Paulus, V. Musil, G. Martius, and M. Rolinek. Differentiation of black-box combinatorial solvers. In *International conference on learning representations*, 2020.

[46] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[47] J. Robinson. On the Hamiltonian game (a traveling salesman problem). Technical report, Rand project air force arlington VA, 1949.

[48] R. T. Rockafellar. Lagrange multipliers and optimality. *SIAM review*, 35(2):183–238, 1993.

[49] M. Rolínek, P. Swoboda, D. Zietlow, A. Paulus, V. Musil, and G. Martius. Deep graph matching via blac-kbox differentiation of combinatorial solvers. In *Computer vision–ECCV 2020: 16th European conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXVIII 16*, pages 407–424. Springer, 2020.

[50] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[51] U. Schmidt and S. Roth. Shrinkage fields for effective image restoration. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2774–2781, 2014.

[52] N. Z. Shor. Quadratic optimization problems. *Soviet journal of computer and systems sciences*, 25:1–11, 1987.

[53] N. Z. Shor. Dual quadratic estimates in polynomial and boolean programming. *Annals of Operations Research*, 25(1):163–168, 1990.

[54] K. Sörensen. Metaheuristics: the metaphor exposed. *International transactions in operational research*, 22(1):3–18, 2015.

[55] D. Sprecher. A representation theorem for continuous functions of several variables. *Proceedings of the American mathematical society*, 16(2):200–203, 1965.

[56] C. Summers and M. J. Dinneen. Nondeterminism and instability in neural network optimization. In *International Conference on Machine Learning*, pages 9913–9922. PMLR, 2021.

[57] K. Weierstrass. Über die analytische darstellbarkeit sogenannter willkürlicher functionen einer reellen veränderlichen. *Sitzungsberichte der königlich preußischen akademie der wissenschaften zu Berlin*, 2:633–639, 1885.

[58] M. Wright. The interior-point revolution in optimization: history, recent developments, and lasting consequences. *Bulletin of the American mathematical society*, 42(1):39–56, 2005.