Continuous Optimization — Algebra Project

Francesc, Mateo, Izan, Èric, Tomi Ockier

February 27, 2024

Contents

1 Introduction

From other courses in our degree, we know that training a machine learning algorithm often is just finding a good set of parameters for that algorithm (Fig 1). In this work, we aim to explain the methods of continuous optimization that make possible to achieve this feat.

We will look at the two main branches of this subject, unconstrained and constrained optimization. The infamous gradient descent falls into the first category, while the Lagrange multipliers and convex optimization belong to the second one. We are going to cover all these topics. The figure 2 is an overview of the topics in the form a graph.

The type of problems that continuous optimization aims to solve are of the following nature, a minimization

$$\min_x f(x), \tag{1}$$

Figure 1: In this illustrative joke, what we call the training/optimization of the machine learning algorithm would be the "stirring of the pile". Source: www.xkcd.com/1838

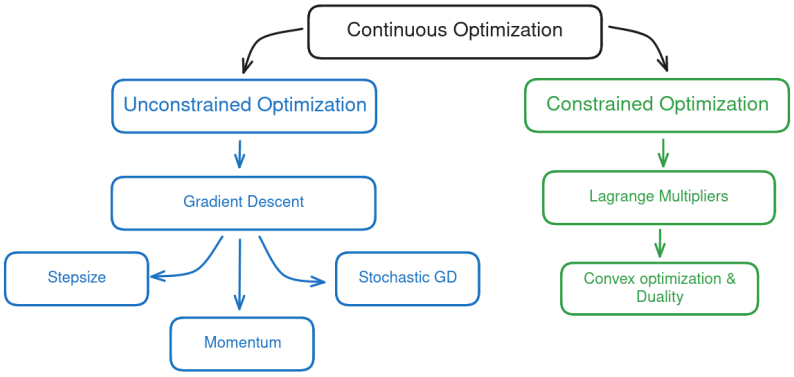

Figure 2: Graph that works as an overview of the subject illustrated in this project. Note that there are two main ideas, gradient descent and Lagrange multipliers, from which we follow the explanations of the over ones.

or a maximization:

$$\max_x f(x), \tag{2}$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is the objective functions that the problem needs to solve. Note that the function returns an integer, this is usually the case in machine learning algorithms, even in categorical classification (see figure 3). The most common case in ML is that the function is minimized, this is a convention.

Taking into account this problem definition, constrained optimization is the special case where the minimum or maximum have to be found inside a determined set of possible values of the function. On the contrary, with unconstrained optimization, the optimized values can be element of the image.

Note that the constrain applies to the possible values; not the parameters of the function, those

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2$$

Figure 3: The Mean Squared Error, here illustrated, is often used in simple ML models. The mean is performed over n data points. It can be also used in categorical classification has it outputs an mean of the results used as examples to trained the model in a supervised ML task. Thus, the overall error of the model is taken into account in the optimization.

can be any value on the domain. As one more note before starting with the main part of the document, if we want to minimize a function f by changing the parameter x, this minimum value is represented as

$$\min_x f(x)$$

Whereas the set of values of x that accomplishes this, is pictured as the set

$$X = \arg\min_x f(x).$$

If it is nonempty, we know the optimization problem is solvable. Consider this an assumption taken for the rest of the document. Alongside it, we will also consider that any objective function is continuously differentiable.

We can also constrain the parameters to be optimized using:

$$\min_{x \in \mathbb{R}^n} f(x)$$

The condition to meet here is that they have to belong to a set.

2 Optimization Using Gradient Descent

The gradient descent method is very important to both the current society and our degree, as it is responsible for nearly all the major advances in AI that we have seen this decade.

It is used in nearly every deep learning model that we know of, alongside the backpropagation method for calculation of the gradients. This algorithms always work with each other regarding deep learning models.

In this section we hope to provide a good explanation how to do continuous optimization using gradient descent.

2.1 Motivation behind gradient descent

From high-school we know that the critical points of a function (including the maximum and the minimum) can be found by the root of the derivative:

$$\frac{\partial f}{\partial x} = 0$$

This implies that an analytical solution has to be found for the derivative. If the same approach is applied to the machine learning models of the last decade, especially for deep learning ones. These solutions are impossible to reach due the large quantity parameters that they depend on. A new method to minimize the objective functions of these models needs to be found.

Gradient descent fits this task perfectly. In the simplest of terms, it can be described as an iterative algorithm, similar to the Newton method for solving the roots of a function. Gradient descent can be considered somewhat of its spiritual successor.

The main idea behind it is to evaluate the negative gradient of a given point and take steps towards its direction. As with Newton's method, it involves consecutive evaluations of a derivative.

More intuitively, with gradient descent we try to flow downhill, just as water does, always trying to reach the lowest point. The change of parameters proportional to the gradient is what allows us to make this downhill trajectory, eventually arriving at the minimum of the function.

2.2 Basic description

The following formula describes the method:

$$x_{i+1} = x_i - \gamma_i(\nabla f)(x_i) \tag{3}$$

Where x_i is the parameter to optimize. The subscript i indicates that the parameter is in its initial state at the beginning of the iteration. x_{i+1} is the parameter at the end of the iteration. The whole updated depends on the gradient $\nabla f(x_i)$ and one hyperparameter, called step-size. It indicates the rate at which the parameters x_i are updated. On the following section we develop further this concept.

The initial parameter can be imagined as a coordinate in a n-dimensional space. The gradient is the specific inclination that the function has at that given point. Basically what we are doing when applying this formula is lowering the value of $f(x_i)$ so that $f(x_1) \geq f(x_{i+1}) \geq f(x_{i+2}) \geq \cdots$, eventually reaching a minimum.

However, this is a naive approach at gradient descent. We might find a situation in which the steps taken are too little or too large. This can lead to a far larger convergence rate than desired. Two simple heuristics that control the step-size can be introduced to mitigate this negative effects:

- When the value of the function decreases, we can conclude that we can advance with larger steps. This will reduce the amount of steps needed to reach the minimum, reaching convergence faster.
- When the value of the function increases after a gradient step (meaning a failed iteration), we can conclude that the step wat too large and we overshoot. When it occurs, the step can be undone and correct the step-size to a smaller value. Thus, when we repeat the iteration the overshot is less likely to occur.

Example 2.1. Example of gradient descent when solving linear equations of the form $Ax = b$. When we try to solve a system of linear equations of this form we are basically trying to solve $Ax - b = 0$ by finding which x minimizes the squared error

$$||\mathbf{A}x - b||^2 = (\mathbf{A}x - \mathbf{A})^T(\mathbf{A}x - \mathbf{A}),$$

using the Euclidean norm. We can get the gradient:

$$\boldsymbol{\nabla x} = 2(\mathbf{A}x - b)^T\mathbf{A}$$

This formula can be used directly in a gradient descent algorithm. Moreover, in this specific type of optimization the convergence speed is dependent on the *condition number* $\kappa = \frac{\sigma(\mathbf{A})_{\max}}{\sigma(\mathbf{A})_{\min}}$. Where $\sigma(\mathbf{A})$ are the singular values of the matrix $\mathbf{A}$. Note that this particular example is an application of the MSE (figure 3)

In terms of machine learning, specially deep learning, the step-size is often called *learning rate*. However, sometimes there is a difference between them, there is not a consensus.

Most often, on the implementations of this methods the learning rate or step-size is a small value, similar to 10^{-3}. Moreover, the operation in equation 3 and its more complicated versions are called *optimizers*. It would be very rare to see it implemented with a larger values than 10^{-3} on current deep learning models.

To summarize, choosing a good step-size is important. Not only we reduce the amount of computation due to a faster convergence, we also can smooth and dampen oscillations present on the optimization.

However we need to mention, that this type of algorithms in practice are implemented in huge Python frameworks for deep learning (e.g. Pytorch, TensorFlow and Jax). There are also light-weight alternatives like TinyGrad. Their implementations are production ready for most cases. When extremely good performance its needed, this frameworks are extended with low level CUDA[1] programming, but more simple techniques can be used. We will explain them shortly.

2.3 Optimizers involving momentum

Rather than the simple method described before, n practice other types of optimizers are used. The most notable and simple is the concept of *momentum* that is introduced into the procedure to get faster convergence.

Imagine we want to reach the optimum point of the function, but as we reduce the step-size we are "bouncing" or clipping off the walls taking more steps than necessary. To solve this, we can implement some memory into the gradient descent method.

An new element $\alpha\Delta x_i$ is added to the equation:

$$x_{i+1} = x_i - \gamma(\nabla f)(x_i) + \alpha\Delta x_i \tag{4}$$

$$\Delta x_i = x_i - x_{i-1} = \alpha\Delta x_{i-1} - \gamma_{i-1}(\nabla f)(x_{i-1}), \tag{5}$$

where α usually is between 0 and 1. It indicates how quickly the contributions of the previous gradient exponentially decay. This is so the momentum keeps "forgetting" the gradient updates that where done several iterations before. From what we understand α determines how "long the memory" of the momentum is.

By doing, so we are taking into account the last iterations so that the gradient descent smooths out. The step size is larger when there have been several iterations where the gradient has pointed out in the same direction. If this happens is logical to assume that we are going in a stable direction that needs to be followed, thus the steps could be larger.

This method is especially useful when we only have an approximation of the gradient as we are going to explain next.

[1] CUDA is the core "architecture" used in Nvidia GPUs, the state of the art in hardware acceleration for AI/DL. They provide a set of toolkits that can be used to optimize has much as possible the implementations of the gradient descent and backpropagation algorithms. However, most of the time with the tools provided in the Python DL frameworks it is enough.

Example 2.2. Simple gradient descent implementation using Sage Math.

```
#Function that we want to optimize
def f(x1,x2):
    return (1/2 * matrix([x1, x2]) * matrix([[2,1], [1,20]]) * matrix([[x1], [x2]])
    - matrix([5,3]) * matrix([[x1], [x2]]))

#Gradient of the function
def gradf(x1,x2):
    return (matrix([x1, x2]) * matrix([[2,1], [1,20]]) - matrix([5,3]))

#Gradient descent
def gradient_descent(x1,x2,step_size):
    Continue = True
    while Continue:
        final_pos = matrix([[x1],[x2]]) - step_size*gradf(x1,x2).transpose()
        f1 = (final_pos[0][0])
        f2 = (final_pos[1][0])

        if float(f(f1,f2)[0][0]) <= float(f(x1,x2)[0][0]):
            x1 = f1
            x2 = f2
        else:
            Continue = False
            return final_pos

print(gradient_descent(-3,-1, 0.085))


```

2.4 Stochastic gradient descent

Due to the gargantuan size of neural networks used in deep learning, some optimization is required when evaluating the gradients of a parameter in the network. Thus, approximations of the gradients are used in practice. To reach the minimum point we do not need to compute the exact gradient, we can still go towards it with an approximation, which is far easier to calculate. This method is called *Stochastic gradient descent* (SGD).

Going back to the figure where MSE was introduced (figure 3, in which the overall objective function is composed of a sum. The same applies to other popular machine learning losses like the log-likelihood used in supervised deep learning:

$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log p(y_n|\boldsymbol{x}, \boldsymbol{\theta})$$

where $\boldsymbol{x}_n \in \mathbb{R}^D$ are vectors that belong to the training set, used as inputs to the network and therefore to the loss. The training labels corresponding to the inputs are y_n, and the parameters of the model $\boldsymbol{\theta}$. The gradient is calculated over the N examples that we take into account. The most common objective or functions for this kind of model have this form, a sum of many errors.

What the SGD method does is just a sample (just a single example) from the training set to perform the optimization on. Only the gradient of one example is computed at each iteration of the optimization.

The opposite method would be batch gradient descent, which takes into account every training example at each iteration. The name is a bit confusing, because by "batch" we expect a group of example. However, this is the case for mini-batch gradient descent that consists of a small selection of training examples to update the parameters.

The most important factor of SGD or mini-batch methods is that the size of the training examples does not matter. If we were to train on all training examples, the convergence time would increase drastically as the training dataset grew larger.

The key idea is that for convergence we require only an estimation of the "true" gradient, which is considered in theory the gradient over all training examples. Thus, in practice an empirical estimate of the expected value of the gradient is used.

However, there is a risk with this approach, as the estimation needs to be unbiased, careful data gathering needs to be used, as well as a random shuffle of the training data. To solve this, all models require a split of the dataset into a training dataset and a test dataset. The latter is used to evaluate the model, to ensure that there is no bias on the training data.

Note that there is always a trade off between the size of the mini-batches and the accuracy of the gradient. Usually an optimized mini-batch size is selected, taking into account the limitations and architecture of the hardware. For the training of this models, highly optimized parallel operations performed on GPUs are used; and they may have an optimal way to use them. The idea is to get a larger as possible batch size, in order to get a more stable convergence while staying between the capabilities of hardware, to achieve a non-expensive calculation. Note that this is, most notably, because the GPUs have a limited amount of memory.

The most common way to optimize the training of deep learning models is to tune the hyperparameters of it. Usually, close to optimal batch sizes and learning rates can be found. The learning rate can be chosen by trial and error, but the best technique to obtained is by monitoring the learning curves that plot the objective function as a function of time[2].

For even more optimization and performance, as we already said, some low-level programming can be used to better fit the hardware accelerator, the GPU. However, before that, some memory management can be applied. This is to leverage better the loading of the training samples from the computer memory or the mass storage (SSDs in this case), as we need to ensure that there no bottlenecks. The images need to be served to the model as fast as possible, which it loads to the memory of the GPU (the VRAM), that is the type of memory that need to be manage more closely.

2.5 Optimizers used in Deep Learning

Combining both the concepts of SGD and Momentum, we are going to explain the most popular adaptive learning rate methods. Notice that a special attention is placed upon this hyperparameter, as it is often very sensible during training. Small changes on the initial learning rate or step-size, when there is not an adaptive method to change it, can have big impacts on the training performance of the model.

2.5.1 Adaptive Gradient Descent (AdaGrad)

Traditional gradient descent may encounter challenges such as excessive steps or "bouncing" off walls due to diminishing step sizes. To address this, AdaGrad introduces a memory component

[2]See the Deep Learning Book for more info.

into the gradient descent algorithm. It takes into account the square root of the sum of all squared value of the gradient.

First we compute the gradient over m training examples taking the average:

$$\boldsymbol{g} = \frac{1}{m}\boldsymbol{\nabla}\mathcal{L}(\cdot) \tag{6}$$

The accumulated gradient is updated by the squared element-wise of the gradient:

$$\boldsymbol{r} = \boldsymbol{r}_i + \boldsymbol{g} \cdot \boldsymbol{g}$$

We introduce the learning rate μ alongside another parameter δ to create the new "momentum":

$$\Delta\boldsymbol{\theta} = -\frac{\mu}{\delta + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{g}$$

Note that the division and the square root are applied element-wise. Finally the parameter of the model x is updated based on the previous equation:

$$x_{i+1} = x_i + \Delta\boldsymbol{\theta}$$

The algorithm considers information from previous iterations, smoothing out the gradient descent process. As we said before, this proves particularly beneficial in situations where the exact gradient is unknown, relying on approximations instead.

2.5.2 Root Mean Squared Propagation (RMSProp)

Building upon the foundation laid by AdaGrad, the RMSProp algorithm further refines the adaptive gradient descent strategy. It addresses the issue of slow convergence in AdaGrad by introducing a decay factor to the historical squared gradients. The gradients are less important the more time ago they were calculated.

We evaluate the gradient $\boldsymbol{g}$ like before, with an average over m training examples. However, now the accumulated gradient is updated with a new hyperparameter ρ that controls the decay:

$$\boldsymbol{r} = \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \cdot \boldsymbol{g}$$

With the parameter update taking place in the same way as AdaGrad:

$$x_{i+1} = x_i - \frac{\mu}{\delta + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{g},$$

again, with element-wise operations.

With RMSProp, the decay rate ensures that older squared gradients have a diminishing impact, preventing the continuous growth observed in AdaGrad. It usually preferred over AdaGrad, as it is an improvement upon that method.

2.5.3 Adaptive Moment Estimation (ADAM)

The Adam optimizer synthesizes the strengths of both momentum-based methods and squared-gradient methods. It is a popular choice in deep learning applications due to its efficiency and empirical success.

With ADAM, two types of momentum are introduced, as well as the decaying rates that they have associated. We can think that one momentum as the velocity and the other one as the acceleration.

Most of this procedures have predefined hyperparameters (e.g. the decaying rates) that usually are not changed from model to model. Their usual values have been found to usually work the best. Most of the time, when creating or training a model, on the actual code the only thing specified regarding optimization is the optimizer (SGD, AdaGrad, Adam...) and not the decaying rates for example.

Refer to the original paper for its implementation, as it is out of the scope for this work.

3 Constrained Optimization and Lagrange Multipliers

From now we will consider the problem detailed at equation 1 with added constrains. As a remainder the constrain is on the possible values that the function can take, not on the parameters that we can change during the minimization. The problem that we are trying to solve is:

$$\min_x f(\boldsymbol{x}) \tag{7}$$

$$\text{subject to } g_i(\boldsymbol{x}) \leq 0 \quad \forall i = 1, \cdots, m \tag{8}$$

Where both $f(\cdot)$ and $g_i(\cdot)$ are functions from $\mathbb{R}^D$ to $\mathbb{R}$, and the input vectors $\boldsymbol{x}$ are of course from $\mathbb{R}^D$.

To solve the problem will introduce a Lagrangian via the *Lagrange multipliers* λ_i, a set of m coefficients that multiply each constrain function $g_i(\cdot)$:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i g_i(\boldsymbol{x}),$$

that can be simplified with a dot product using between vectors $\boldsymbol{\lambda}, \boldsymbol{g}(\boldsymbol{x}) \in \mathbb{R}^m$:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x}). \tag{9}$$

We will use this representation of the problem from now on.

Going to back the problem definition, now it must be defined with the added constrains $g_i(x)$:

$$\min_x \quad f(\boldsymbol{x}) \tag{10}$$

$$\text{subject to} \quad g_i(\boldsymbol{x}) \leq 0 \quad \forall i = 1, \cdots, m \tag{11}$$

How, if the take the gradients of the lagrangian, equal them to zero and solve; we get the critical points of the optimization problem with the constrains taken into account.

On a more intuitive level what the lagrangian does is add variables to the problem. It allows us to "move" the constrain wherever we want during the optimization, this is done to find a point where the counter line of $f(x)$ that is tangent to the constrain. This are the stationary points that correspond to the lagrangian.

Example 3.1. Constrained optimization example where we optimize a the plane $f(x, y) = x + y$ with the constrain $g(x, y) = x^2 + y^2 - 1$ which is the unit circle. This example was also shown in our oral presentation. There is a more intricate example in figure 4

```
# Define variables and the function
var('x y l')
f = x+y
g = x^2 + y^2 - 1

# Define the Lagrangian function
L = f - l * g

# Calculate the partial derivatives
dL_dx = diff(L, x)
dL_dy = diff(L, y)
dL_dl = diff(L, l)

# Solve the system of equations to find critical points
solutions = solve([dL_dx == 0, dL_dy == 0, dL_dl == 0], x, y, l, solution_dict=True)

# Print the results
for solution in solutions:
    print("Critical point:", solution)
    print("Value of the function at the critical point:", f.subs(solution))

```

3.1 Uses and implementation

The most notable use case for lagrange multiplier that is closely related to our Artificial Intelligence degree is their use in constrained reinforcement learning. The problem consists on the control in the behavior of single or multiple agents. They learn by a reward function that encapsulates the definition of the problem (e.g. maximize the points obtained on a videogame). However, it is common to not perfectly define the reward function, thus sometimes constrains have to be implemented.

Of course this constrains are not directly on the reward function, they are on the parameters to optimize. In the RL case would be the ones that control the behavior of the agent, their actions.

It seems that their implementation on these types of problems can solve some overshoot and oscillations that can happen on a constrained RL model. Both these problems have already been explained on the unconstrained optimization gradient.

We provide two simple example of lagrangian multiplier with their implementations using Sage Math (4 and 3.1). We also recommend the use of Scipy and other related Python libraries to implement this type of problems because they usually have more features than Sage Math, which stronghold is easy of use compared to pure Python (which is also very easy).

3.2 Primal and Dual problem

For the next section we need to define two types of problems. The first one is know as the *primal problem* which are the previous equations (10). We would say that this problem deals with the primal variables x, which in this case is the vector $\boldsymbol{x}$.

(a) Code implementing the optimization. We directly calculate the gradient and then tell Sage to solve when they equal zero. We need to tell which parameters need to change to both calculate the gradient and solve the equations. This implies that multiple constrained can also be calculated in Sage Math, however it is out of scope for this project.

```
# Define variables and the function
var('x y l')
f = x^2 * y
g = x^2 + y^2 - 3

# Define the Lagrangian function
L = f - l * g

# Calculate the partial derivatives
dL_dx = diff(L, x)
dL_dy = diff(L, y)
dL_dl = diff(L, l)

# Solve the system of equations to find critical points
solutions = solve([dL_dx == 0, dL_dy == 0, dL_dl == 0], x, y, l, solution_dict=True)

Print the results
for solution in solutions:
    print("Critical point:", solution)
    print("Value of the function at the critical point:", f.subs(solution))

```

(b) The output of the code. There is a total of 6 critical points.

```
Critical point: {x: 0, y: sqrt(3), l: 0}
Value of the function at the critical point: 0


Critical point: {x: 0, y: -sqrt(3), l: 0}
Value of the function at the critical point: 0


Critical point: {x: sqrt(2), y: 1, l: 1}
Value of the function at the critical point: 2


Critical point: {x: sqrt(2), y: -1, l: -1}
Value of the function at the critical point: -2


Critical point: {x: -sqrt(2), y: 1, l: 1}
Value of the function at the critical point: 2


Critical point: {x: -sqrt(2), y: -1, l: -1}
Value of the function at the critical point: -2
```

(c) Graph where the optimization problem is pictured.

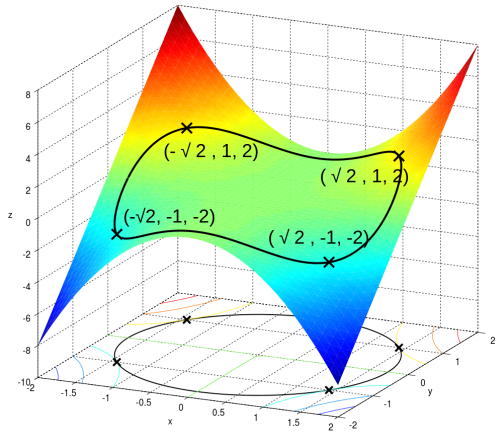

Figure 4: Constrained optimization example where we optimize a the function $f(x, y) = x^2 y$ with the constrain $g(x, y) = x^2 + y^2 - 3$ which is also a circle. Here the results are a total of 6 critical points. With a total of two minimums and maximums, with -2 and 2, values respectively. There is also two points with are saddle points (inflection point that has a value of 0). Note how the constrain is pictured on the bottom of the graph as a circle.

To the primal problem there is an associated *Lagrangian dual problem* that in this case is an optimization over the lagrangian multipliers:

$$\max_{\boldsymbol{\lambda} \in \mathbb{R}^m} \quad \mathcal{D}(\boldsymbol{\lambda}) \tag{12}$$

$$\text{subject to} \quad \boldsymbol{\lambda} \geq \mathbf{0} \tag{13}$$

The function that is maximized $\boldsymbol{\lambda}$ is equal to the minimization of the lagrangian already defined in equation 9. Thus, we end up with a maximization of a minimization:

$$\max_{\boldsymbol{\lambda} \in \mathbb{R}^m} \quad \mathcal{D}(\boldsymbol{\lambda}) = \max_{\boldsymbol{\lambda} \in \mathbb{R}^m} \min_{\boldsymbol{x} \in \mathbb{R}^d} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) \tag{14}$$

$$\text{subject to} \quad \boldsymbol{\lambda} \geq \mathbf{0} \tag{15}$$

From this two problems, the primal and the dual, we can derive various concepts that we are going to explain in the next section. By convention, the primal is not minimized nor maximized. The only thing to take into account is that if one is minimized the other one must be maximized. They cannot behave the same way.

When we encounter a minimax or maximin, the *minimax inequality* stipulates that the maximin is less or equal to the minimax:

$$\max_y \min_y \varphi(\boldsymbol{x}, \boldsymbol{y}) \leq \min_x \max_y \varphi(\boldsymbol{x}, \boldsymbol{y})$$

Putting our problem's terms into the inequality we get that:

$$\min_{\boldsymbol{x} \in \mathbb{R}^d} \max_{\boldsymbol{\lambda} \geq 0} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) \geq \max_{\boldsymbol{\lambda} \geq 0} \min_{x \in \mathbb{R}^d} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$$

The right hand part of the equation is the definition of the dual problem seen on equations 14 and 14. The result of this inequality for this particular problem is known as *weak duality*. The key insight is that the primal values are always greater or equal to the dual values. This information can be used to our advantage as we can choose which problem to optimize based on what is more easy and computational inexpensive. Furthermore, now $\min_{\boldsymbol{x}} \boldsymbol{x}, \boldsymbol{\lambda}$ is an constrained optimization for discrete values of λ. If solve it is easy, then the initial constrained problem is easy to solve.

4 Convex constrained optimization

When the functions on the lagrangian optimization problem are convex, the problem behaves in a different way that allows us to solve it more easily. In this section we are going to look at this advantages.

A *convex optimization problem* is one such that:

$$\min_x \boldsymbol{x} f(\boldsymbol{x}) \tag{16}$$

$$\text{subject to } g_i(\boldsymbol{x}) \leq 0 \quad \forall i = 1, \cdots, m \tag{17}$$

$$h_j(\boldsymbol{x}) = 0 \quad \forall j = 1, \cdots, m \tag{18}$$

where all the functions $f(\boldsymbol{x})$ and $g_i(\boldsymbol{x})$ are convex functions and all $h_j(\boldsymbol{x}) = 0$ are all convex sets.

To solve this types of problems, we can procedure with the lagrange multipliers like before. What it is interesting is the results that we get, because this time, the problem has *strong duality*. Which means that the solutions to the primal and dual problem have the same values, the maximin inequality does not hold.

Therefore, we can choose freely which function to optimize, the dual problem or the primal. Also that because of their nature one is a maximization and the other one a minimization. We can also say that convex functions are in general easier to optimize that concave ones.

There are two particular kinds of convex optimization that we are going to look, linear and quadratic programming.

4.1 Linear and Quadratic programming

A problem where all the functions are linear:

$$\begin{aligned} &\min_{\boldsymbol{x}\in\mathbb{R}^d} \boldsymbol{c}^T\boldsymbol{x} \\ &\text{subject to } \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \end{aligned}$$

Where the dimensions of $\boldsymbol{A}$ is $m \times d$ and it follow that $\boldsymbol{b}$ is m dimension vector. Thus, it has d variables and m linear constrains. Because we need the constrain to equal zero, we include it has $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$ in the lagrangian:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\lambda}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}),$$

the lagrangian multipliers are introduced in the form of the vector $\boldsymbol{\lambda} \in \mathbb{R}^m$ that is non-negative.

From the lagrangian we can form another lagrangian for the dual problem:

$$\mathcal{D}(\boldsymbol{\lambda}) = -\boldsymbol{b}^T\boldsymbol{\lambda},$$

that to subjected to the constrains $\boldsymbol{c} + \boldsymbol{A}\boldsymbol{\lambda} = \boldsymbol{0}$ and $\boldsymbol{\lambda} \geq \boldsymbol{0}$ would be maximized to solve the problem.

Now that we have a dual and primal problem we can choose which to optimize. This is because they have all complex functions. In this particular case since $\boldsymbol{\lambda} \in \mathbb{R}^2$ and $\boldsymbol{x} \in \mathbb{R}^m$, we can choose the problem in the basis of which has the least number of variables. Recall that d is the number of variables and m the number of constrains.

How consider the following quadratic convex optimization problem:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^d} \quad & \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \end{aligned}$$

Note that in this case the constrains are affine, they do not equal zero. This is a *quadratic problem*. Like the linear one, it has d variables and m constrains since $A \in \mathbb{R}^{m\times d}$ and $Q \in \mathbb{R}^{d\times d}$.

Introducing the lagrangian multipliers to redefine the problem we get:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + (\boldsymbol{c} + \boldsymbol{A}^T\boldsymbol{\lambda})^T\boldsymbol{x} - \boldsymbol{\lambda}^T\boldsymbol{b}$$

By taking the derivative with respect to $\boldsymbol{x}$ and make it equal to zero, it is possible to rearrange to isolate the variables:

$$\boldsymbol{x} = -\boldsymbol{Q}^{-1}(\boldsymbol{c} + \boldsymbol{A}^T\boldsymbol{\lambda})$$

By doing so, we can take them out of the lagrangian completely to define the dual problem just with respect to the lagrangian multipliers:

$$\mathcal{D}(\boldsymbol{\lambda}) = -\frac{1}{2}(\boldsymbol{c} + \boldsymbol{A}^T\boldsymbol{\lambda})^T\boldsymbol{Q}^{-1}(\boldsymbol{c} + \boldsymbol{A}^T\boldsymbol{\lambda}) - \boldsymbol{\lambda}^T\boldsymbol{b}$$

Thus, once again we can choose between the primal and dual problem. Just like we need with the linear case, we can choose based on the number of variables compared to the number of constrains.

4.2 Legendre–Fenchel Transform and Convex Conjugate

The final section of the book was not included in our oral presentation, therefore we are going to summarize it.

The *Legendre transform* can be used to describe convex functions by their gradients. This is because, the transform, what really does is describe a convex set or function by their supporting hyperplanes. This is relevant since the hyperplane just touches the function at some points. They do not interact in any other way. Thus, it is tangent to it, the gradient and the hyperplane are equivalent.

By describing the function by its gradient, we can easily define dual and primal problems. This information can help us in machine learning since we can apply this to a convex loss or objective function.