Alma Mater Studiorum · University of Bologna

Computer Science and Engineering department

Master degree in Artificial Intelligence

Empirical Model Learning for Constrained Black Box Optimization

Supervisor:
Michele Lombardi

Co-supervisor:
Andrea Borghesi

Presented by:
Daniele Verì

Session 5

Academic year 2020/2021

Contents

Chapter 1

Introduction

Black box optimization is a field of the global optimization which consists in a family of methods intended to minimize or maximize an objective function that doesn't allow the exploitation of gradients, linearity or convexity information. Beside that the objective is often a problem that requires a significant amount of time/resources to query a point and thus the goal is to go as close as possible to the optimum with the less number of iterations possible.

The key elements of these methods are the usage of a surrogate generator that fit an interpretable mathematical model on the data points and the exploration/exploitation trade off that drive the search in an intelligent way.

In the late '70s it was invented the Bayesian Optimization that became de-facto the State Of The Art in BBO. However the research keep pushing to find methods capable of work with more dimensions, more data points and especially be able to deal with constraints.

The Emprical Model Learning is a methodology for merging Machine Learning and optimization techniques like Constraint Programming and Mixed Integer Linear Programming by extracting decision models from the data.

The learned surrogate model is embedded into a classical optimization framework in a passive way, meaning that the prescriptive model in a setup where all the samples has been already gathered. This allows to exploit both the ability to model high dimensional complex events of NN and the expressiveness of constraint based models.

This work aims to close the gap between Empirical Model Learning optimization and Black Box optimization methods (which have a strong literature) via active learning.

At each iteration of the optimization loop a ML model is fitted on the data points and it is embedded in a prescriptive model using the EML.
The encoded model is then enriched with domain specific constraints and is optimized selecting the next point to query and add to the collection of samples.

The main advantage of this approach w.r.t other BBO methods is that allow an easy integration with complex constraint and even combinatorial ones, with the guarantee that will be always sampled points in the feasible region.

In the following chapters will be firstly introduced the technical background of both BBO and EML, then will be described the functioning of the optimization loop, from the general idea to the implementation challenges that required several engineering techniques to overcome.

In the last chapter we show how the algorithm performed on a known test problem for optimization, compared to other methods from related works.

Finally we present the results on practical real world problem called neural network quantization which is a technique to reduce the memory footprint of neural models that is attracting more and more attention especially for its use in edge computing applications.

Chapter 2

Background

2.1 Black Box Optimization

Global optimization is a branch of applied mathematics and numerical analysis that attempts to find the global minima or maxima of a function on a given set. [1]

Among the many methods existing in literature, black box optimization is the one that makes the least assumptions about the function properties, like the linearity, the convexity and so on. For this reason is a topic of critical importance in many areas including complex systems engineering, energy and the environment, materials design, drug discovery, chemical process synthesis, and computational biology.

2.2 Bayesian optimization

Bayesian optimization is a sequential design strategy for global optimization of black-box functions that does not assume any functional forms. It is usually employed to optimize expensive-to-evaluate functions.

Bayesian optimization is typically used on problems of the form $\max_{x \in A} f(x)$, where A is a set of points whose membership can easily be evaluated. Bayesian optimization is particularly advantageous for problems where $f(x)$ is difficult to evaluate, is a black box with some unknown structure and where derivatives are not evaluated. [2]

The Bayesian strategy treats the objective as a random function and place a prior over it. The prior captures beliefs about the behavior of the function. After gathering the function evaluations, the prior is updated to form the posterior distribution over the objective function. In practice this is done with the Gaussian Process, a machine learning model that perform better if the domain knowledge (prior) in properly integrated.

The posterior distribution, in turn, is used to construct an acquisition function that determines the next query point balancing exploration and exploitation, with the goal of minimize the expensive function evaluations.

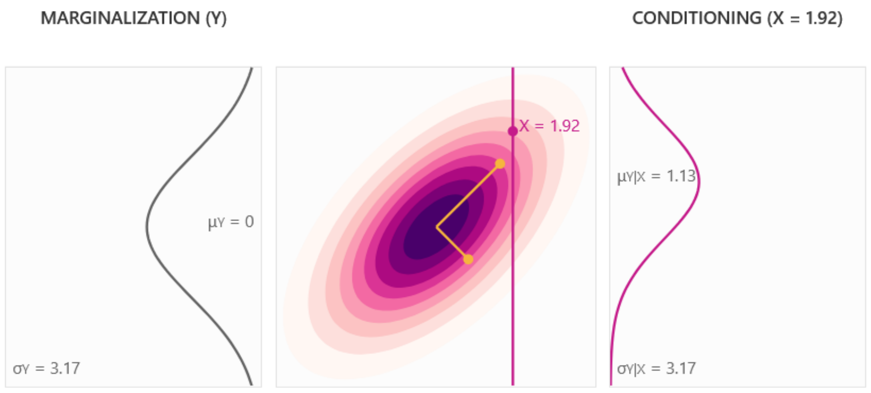


Figure 2.1: Marginalization and conditioning [3]

2.2.1 Gaussian Process

Gaussian processes are a powerful tool in the machine learning toolbox. They allow to make predictions about the data by incorporating prior knowledge. Their most obvious area of application is fitting a function to the data. This is called regression and is used, for example, in robotics or time series forecasting.

For a given set of training points, there are potentially infinitely many functions that fit the data.
Gaussian processes offer an elegant solution to this problem by assigning a probability to each of these functions.
The mean of this probability distribution then represents the most probable characterization of the data. Furthermore, using a probabilistic approach allows to incorporate the confidence of the prediction into the regression result. [3]

The multivariate Gaussian distribution is the basic building block of Gaussian process. It is defined by a mean vector μ and a covariance matrix Σ.
The mean vector describes the expected value of the distribution. Each of its components describes the mean of the corresponding dimension. Σ models the variance along each dimension and determines how the different random variables are correlated. The covariance matrix is always symmetric and positive semi-definite. The diagonal of Σ consists of the variance σ_i^2 of the i-th random variable. And the off-diagonal elements σ_{ij} describe the correlation between the i-th and j-th random variable.

The Gausssian process computes conditional probabilities and this operation take the name of conditioning: it is used to determine the probability of one variable depending on another variable. This operation is also closed and yields a modified Gaussian distribution. This operation is the cornerstone of Gaussian processes since it allows Bayesian inference. fig.2.1

Stochastic processes, such as Gaussian processes, are essentially a set of random variables. In addition, each of these random variables has a corresponding index i. We will use this index to refer to the i-th dimension of our n-dimensional multivariate distributions.

The goal of Gaussian processes is to learn this underlying distribution from training data. Respective to the test data X, we will denote the training data as Y. As we have mentioned before, the key idea of Gaussian processes is to model the underlying distribution of X together with Y as a multivariate normal distribution. That means that the jointprobability distribution $P_{X|Y}$ spans the space of possible function values for the function that we want to predict. Please note that this joint distribution of test and training data has $|X| + |Y|$ dimensions.

In order to perform regression on the training data, we will treat this problem as Bayesian inference. The essential idea of Bayesian inference is to update the current hypothesis as new information becomes available. In the case of Gaussian processes, this information is the training data. Thus, we are interested in the conditional probability $P_{X|Y}$. In Gaussian processes we treat each test point as a random variable. A multivariate Gaussian distribution has the same number of dimensions as the number of random variables.

Now that we have the basic framework of Gaussian processes together, there is only one thing missing: how do we set up this distribution and define the mean μ and the covariance matrix Σ. The covariance matrix Σ is determined by its covariance function k, which is often also called the kernel of the Gaussian process.

$$\Sigma = \begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & \dots & K(x_1, x_n) \\ K(x_2, x_1) & K(x_2, x_2) & \dots & K(x_2, x_n) \\ \vdots & \vdots & \vdots & \vdots \\ K(x_n, x_1) & K(x_n, x_2) & \dots & K(x_n, x_n) \end{bmatrix} \tag{2.1}$$

The kernel functions allow to inject the prior knowledge of the problem to the model.
In practice, to define a kernel we have to hand-pick a parametrized kernel function $K_\theta(x_i, x_j)$ where θ is the parameter vector, collect training observations Y and then choose θ so as to maximize the likelihood of the training data (MLE): $argmax_\theta f(Y, \theta)$ where $f(Y, \theta)$ is the estimated probability density of the observations.
The training problem is a numerical optimization problem solved with local optimality methods like first and second order (LBFG) gradient descend.

In Gaussian processes it is often assumed that $\mu = 0$, which simplifies the necessary equations for conditioning, in fact in practice inputs are normalized to achieve 0 centered distribution.

The entry Σ_{ij} describes how much influence the i-th and j-th point have on each other.

Kernels can be separated into stationary and non-stationary kernels. Stationary kernels, such as the RBF kernel or the periodic kernel, are functions invariant to translations, and the covariance of two points is only dependent on their relative position. Non-stationary kernels, such as the linear kernel, do not have this constraint and depend on an absolute location. fig.2.2

Several kernel can be combined. Usually the Matérn kernel is used in BO.

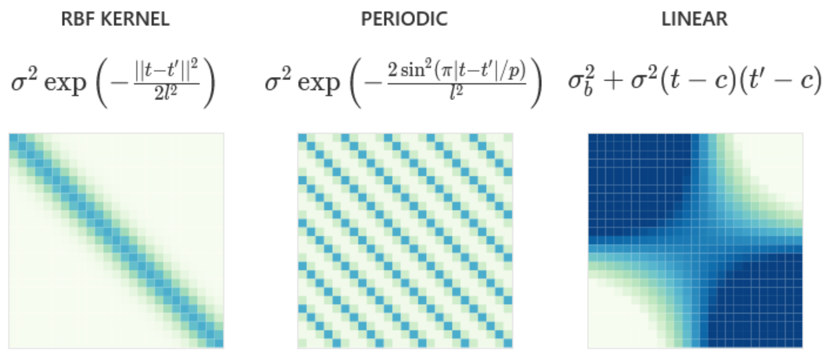


Figure 2.2: Common kernels [3]

2.2.2 Acquisition functions

Acquisition functions are crucial to Bayesian Optimization, they are used to balance exploration and exploitation and there are a wide variety of options. [4] The following acquisition functions are assuming that the optimization is looking for the maximum, but it is straightforward to adapt the formula for minimum problems.

Upper Confidence Bound (UCB)
A trivial acquisition function that combines the exploration/exploitation tradeoff in a linear combination of the mean and the uncertainty of the surrogate model.

$$\alpha(x) = \mu(x) + \lambda\sigma(x) \tag{2.2}$$

The intuition behind the UCB acquisition function is weighing of the importance between the surrogate's mean vs. the surrogate's uncertainty. The λ above is the hyperparameter that can control the preference between exploitation or exploration.

Probability of Improvement (PI)
This acquisition function chooses the next query point as the one which has the highest probability of improvement over the current max $f(x^+)$.

$$x_{t+1} = \operatorname{argmax} \alpha_{PI}(x) = \operatorname{argmax}(P(f(x) \geq (f(x^+) + \epsilon))) \tag{2.3}$$

We are just finding the upper tail probability of the surrogate posterior, but since the conditional probability of a multivariate Gaussian distribution still is a Gaussian distribution, the formula reduce to:

$$x_{t+1} = \underset{x}{\operatorname{argmax}}\, \Phi\left(\frac{\mu_t(x) - f(x^+) - \epsilon}{\sigma_t(x)}\right) \tag{2.4}$$

where Φ is the Gaussian CDF.
PI uses ϵ to strike a balance between exploration and exploitation. Increasing ϵ results in querying locations with a larger σ as their probability density is spread.

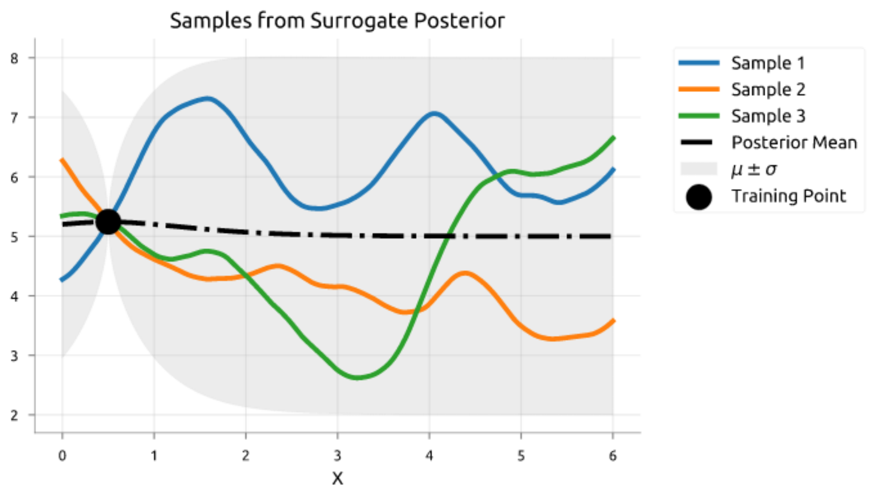


Figure 2.3: Thompson Sampling [4]

Expected Improvement (EI)
Probability of improvement only looked at how likely is an improvement, but, did not consider how much we can improve. The Expected Improvement, does exactly that:

$$x_{t+1} = \underset{x}{\operatorname{argmax}}\, \mathbb{E}(max\{0, h_{t+1}(x) - f(x^+)\}|\mathcal{D}_t) \tag{2.5}$$

where $f(x^+)$ is the maximum value that has been computed so far. This equation for GP surrogates has this analytical expression 2.6:

$$\begin{aligned} EI(x) &= \begin{cases} (\mu_t(x) - f(x^+) - \epsilon)\Phi(Z) + \sigma_t(x)\phi(Z) & \text{if } \sigma_t(x) > 0 \\ 0 & \text{if } \sigma_t(x) = 0 \end{cases} \\ Z &= \frac{\mu_t(x) - f(x^+) - \epsilon}{\sigma_t(x)} \end{aligned} \tag{2.6}$$

where Φ is the Gaussian CDF and ϕ is the PDF. We can see that EI is high when the expected value of $\mu_t(x) - f(x^+)$ is high r the uncertainty $\sigma_t(x)$ around a point is high.ϵ can be used to moderate the amount of exploration/exploitation

Thompson Sampling (TS)
At every step, we sample a function from the surrogate's posterior and optimize it (fig. 2.3). Locations with high uncertainty will show a large variance in the functional values sampled from the surrogate posterior. Thus, there is a non-trivial probability that a sample can take high value in a highly uncertain region. Optimizing such samples can aid exploration.
The sampled functions must pass through the current max value, as there is no uncertainty at the evaluated locations. Thus, optimizing samples from the surrogate posterior will ensure exploiting behavior.

2.3 Empirical Model Learning

Empirical Model Learning (EML) is a methodology that relies on Machine Learning for obtaining components of a prescriptive model, using data either extracted from a predictive model or harvested from a real system.
It consists in embedding Machine Learning models (namely Decision Trees and Artificial Neural Networks) into several optimization techniques like Local Search, Mixed Integer Non-Linear Programming, Constraint Programming, SAT Modulo Theories. [5]

While most works on surrogate models focus on a specific class of Machine Learning techniques, specific problems (typically with continuous variables and range constraints) and solution techniques, e.g. genetic algorithms and derivative-free optimization, EML aims at enabling the use of as many Machine Learning techniques as possible, within as many optimization methods as possible.
The goal is having the ability to choose the most adequate solution approach for solving optimization problems defined over high-complexity systems, which typically have the structure presented in equation 2.7.

$$\begin{aligned} \min \quad & f(x,z) & & \qquad (2.7)\\ \text{s.t.} \quad & g_j(x,z) & \forall j \in J & \\ & z = h(x) & & \\ & x_j \in D_i & \forall x_i \in x & \end{aligned}$$

Where x is the vector inputs with domain D, z is the vector of observable variables related to the target system, while the cost function f may depend on both of them. Problem variables may be subject to domain constraints, represented as logical predicates $g_j(x,z)$. The predicates may correspond to classical inequalities from Mathematical Programming, or to combinatorial restrictions, such as Global Constraints in Constraint Programming (e.g. alldiff, element).

The $h(x)$ function describes the (approximate) behavior of the high-complexity system and specifies how the observable variables depend on the decision variables. The function h corresponds to the encoding of the Empirical Model, obtained via Machine Learning. A surrogate model takes the form of a function with pre-defined structure and unknown parameters

Designing an optimization approach based on EML requires to take care of three main activities:

1. Defining the core combinatorial structure of the problem.
2. Obtaining a Machine Learning model.
3. Embedding the Empirical Model in the combinatorial problem

In this work we focus on embedding a neural network as surrogate models in a MILP prescriptive model.

2.3.1 MILP embedding of neural networks

Mixed Integer Non-Linear Programming is a field of Mathematical Programming that is concerned with finding extreme points of non-linear functions subject to linear, non-linear, or integrality constraints. Modern MINLP solvers can take advantage of the problem structure (constraints and cost function) via convex envelope approximation, linearization, cutting planes, constraint propagation, and branching.

A NN can be embedded in a MINLP model by introducing variables to model the input and output of each neuron (eq. 2.8), and then by directly inserting the neuron equations in the model. This is easy as long as the considered MINLP solver supports the activation functions employed in the neurons. Once the empirical model has been encoded, its equations will be automatically taken it into account by the solver for computing bounds and generating cuts.

$$y = \phi\left(b + \sum_i w_i x_i\right) \tag{2.8}$$

The NN used in this work are composed by just feed forward layers with ReLU activation, in fact being the black box optimization a generic task, there is no prior knowledge to guide the model selection towards architectures like CNN or attention based nets.
The ReLU activation function can be easily encoded in a MILP model because is composed by two linear component: the relation $y = max(0, wx + b)$ can be translated to eq. 2.9.

$$\begin{aligned} & y - s = wx + b \\ & z = 1 \implies s \leq 0 \\ & z = 0 \implies y \leq 0 \\ & y, s \geq 0, x \in \mathbb{R}^n, z \in \{0, 1\} \end{aligned} \tag{2.9}$$

Where s is a slack variable that can assume the value of 0 or y, z is a binary variable and the implication can be linearized with the big-M formulation [6]: $z = 0 \implies y \leq 0$ becomes $y - M \cdot z \leq 0$ with M a constant bigger than the y upper bound.

2.3.2 Branch and Bound

Here follows a breif description of the algorithm ath the core of the MILP solutio process.

Branch-and-bound uses intelligent enumeration to arrive at an optimal solution for a mixed integer program or any special case thereof. This involves construction of a search tree. Each node in the tree consists of the original constraints along with some additional constraints on the bounds of the integer variables to induce those variables to assume integer values. [7]

At each node of the branch-and-bound tree, the algorithm solves a linear programming relaxation of the restricted problem, i.e., the MIP with all its variables relaxed to be continuous. Here we have 3 possibilities:

- The subproblem is optimal with all variables in assuming integer values. In this case, the algorithm can update its best integer feasible solution; this update tightens the upper bound on the optimal objective value.
- The subproblem is infeasible. In this case, no additional branching can restore feasibility and the node is fathomed.
- The subproblem has an optimal solution, but with some of the integer restricted variables assuming fractional values: if the optimal node LP objective is no better than the previously established upper bound on the optimal objective, the node is fathomed; else the algorithm processes the node by creating two child nodes and their associated subproblems with a restriction on the integer variable.

2.4 Applications

So far we described the main concept behind this work. In this section we describe some real world application of BBO.

Black Box Optimization considers the design and analysis of algorithms for problems where the structure of the objective function and/or the constraints defining the set is unknown, unexploitable, or non-existent. The most frequent BBO situation arises when the evaluation of the objective involve the execution of a computer simulation. In BBO problems, functions are often non-differentiable; [8]

The simulation or the process returning the objective and constraint functions may be time-consuming. For example, automotive valve train design requires seconds, sample size identification or studies in the pharmaceutical industry requires minutes, hyperparameter optimization and neural architecture search requires hours, and airfoil trailing-edge noise reduction requires days of CPU time for each execution of the associated computer code or simulation. [8]

The survey in fig. 2.4 contains a large selection of applications where the Mads algorithm (mesh adaptive direct search, a diffused algorithm for it's simplicity and success) was used during the last twenty years; this selection is far from being exhaustive, but highlights different areas in which it was successfully applied. Active learning methods are usually applied in situations where can be executed multiple experiments, often via simulation, or in situation like neural architecture search (NAS).
EML optimization produce always feasible solutions, making it applicable also in situation were constraints satisfaction is critical.
In this work we apply the EML optimization loop to a real world scenario called model quantization.

2.5 Related works in constrained optimization

The plain Bayesian Optimization despite is widely used in unconstrained setups, becomes rapidly not suitable when dealing with constraints.

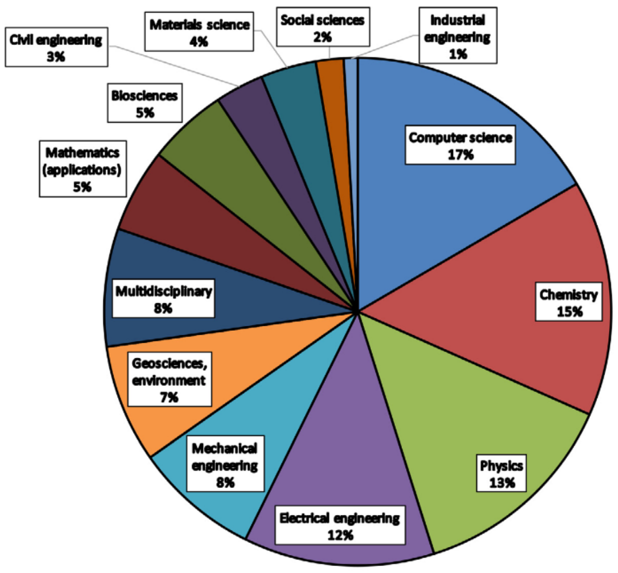


Figure 2.4: Distribution of the application fields that cite the MADS algorithm [8]

Constraints can be known or unknown (Black Box constraints). In this work will be tackled known constraints because we can rely on the expressiveness of CP or MILP solver to integrate complex ones, while for Black Box constraints the system feedback includes also the constraint violation term in addition to the objective value.

The global optimization of a high-dimensional black-box function under black-box constraints is a pervasive task in machine learning, control, and engineering. These problems are challenging since the feasible set is typically non-convex and hard to find, in addition to the curses of dimensionality and the heterogeneity of the underlying functions. In particular, these characteristics dramatically impact the performance of Bayesian optimization methods, that otherwise have become the defacto standard for sample-efficient optimization in unconstrained settings. [9]

Following are listed the main algorithms in the field of constraint black box optimization and their main features.

- **c-EI** [10]: constraint-Expected Improvement extends the expected improvement criterion (EI) to constraints,the expected constrained improvement acquisition function is precisely the standard expected improvement of x over the best feasible point so far weighted by the probability that x is feasible.
- **PESC** [11]: Predictive Entropy Search is a method based on Bayesian Optimization where the acquisition function aims to maximize the expected gain of information on the posterior distribution of the global maximizer. Hern andez-Lobato et al. extended PES to constraints and detailed how to make the sophisticated approximation of the entropy reduction computationally tractable in practice. Their PESC algorithm usually achieves great results and is widely considered the state-of-the-art for constrained BO despite its rather large computational costs.

- **COBYLA** [12]: It works by iteratively approximating the actual constrained optimization problem with linear programming problems. During an iteration, an approximating linear programming problem is solved to obtain a candidate for the optimal solution. The candidate solution is evaluated using the original objective and constraint functions, yielding a new data point in the optimization space.
- **CMA-ES** [13]: is one of the most powerful and versatile evolutionary strategies. It uses a covariance adaptation strategy to learn a second order model of the objective function. CMA-ES handles constraints by the "death penalty" that sets the fitness value of infeasible solutions to zero.
- **SLACK** [14]: By lifting constraints into the objective via the Lagrangian relaxation, Gramacy et al. took a different approach. Note that it results in a series of unconstrained optimization problems that are solved by vanilla BO. SLACK of Picheny et al. refined this idea by introducing slack variables and showed that this augmented Lagrangian achieves a better performance for equality constraints.
- **SCBO** [9]: A new method called Scalable Constrained Bayesian Optimization that is based on three key contributions:
    1. extending Thompson Sampling to constrained optimization;
    2. transforming objective and constraints to simplify the detection of optima and feasible areas;
    3. employ trust regions, that force sampling in local regions, mitigating the problem with typical acquisition functions in high-dimensional contexts, which fail to zoom in on good solutions.

  The algorithm evaluates an initial set of points and initialize the trust region at a point of maximum utility.
  At each iteration, fit GP models to the transformed observations, generate r candidate points in the trust region and for each of the points of the next batch we sample a realization from the posterior over each candidate and add a point of maximum utility to the batch. Finally evaluate the objective and constraints at the q new points, then adapt the trust region by moving it.

It is worth to mention the paper "Constrained Black Box optimization using Mixed Integer Programming" [2021] [15], that was published during the development of this work.
It shares the same main ideas employed in the EML optimization loop except that it is limited to a simple acquisition function which doesn't take into account the uncertainty of the model in unexplored regions.

Chapter 3

EML optimization loop

3.1 Overview

In this chapter will be presented the general structure of the algorithm as well as the implementation challenges that we dealt with.

In the algorithm 1 is represented the general view of the EML optimization loop.

Algorithm 1 EML optimization loop

```
$problem \leftarrow Problem(objective, constraints, bounds, types)$
$x, y \leftarrow problem.sample(initial\_points)$
$i \leftarrow 1$
while $i \leq iterations$ do
    $model \leftarrow fit\_model(x, y)$
    $encoded \leftarrow eml\_encode(model)$
    $opt\_x \leftarrow optimize\_acquisition\_function(encoded)$
    $x \leftarrow concatenate(x, opt\_x)$
    $y \leftarrow concatenate(y, opt\_y)$
    $i \leftarrow i + 1$
end while
$best\_x \leftarrow x[\mathrm{argmin}(y)]$
return $best\_x$
```

We define a "problem" as the combination of objective function, constraints, type and bounds of the variables.

The problem also act as a proxy respect to the invocation of the function and it is responsible to generate the initial points before start the search.
This step is straightforward for unconstrained problems but becomes challenging when dealing with constraints and integer variables. The detailed description of the procedure is postponed to the end of the chapter because it rely on considerations about the other components.

In each iteration the surrogate model is fitted from scratch on the samples and then is embedded in a MILP formulation using the Empirical Model Learning.
The MILP model is then enriched with the specific problem constraints and is solved optimizing an objective function that represent an acquisition function borrowed from Bayesian Optimization methods.

The objective function is queried with the input suggested by the solution of the MILP model. The new data-point is added to collection and the loop is repeated again until the maximum number of iterations is reached.

3.1.1 Domain specific constraints

Most of the related works on constrained BBO handle the constraints in a Black Box fashion: the constraints violation along with the objective value are evaluated at each iteration and the algorithm attempt to minimize both the values.
In our case constraints are known and are integrated directly in the model. This avoid infeasible solution but requires a perfect knowledge of the system.

Choosing to rely on the MILP framework allows to implement convex constraints but also combinatorial ones. Since some solvers (like CPLEX that we used in this work) supports also Quadratic Programming it is possible to integrate even more expressive constraints like $L2$ distance and multiplicative relations (MIQCP).

Despite the EML formulation (2.7) we choose to not handle the constraints in the form $g(x, z)$ but only $g(x)$, meaning that constraints can only be expressed on the decision variables and not on the objective value. This choice is motivated by the fact that especially in the first step of the optimization, the surrogate model can be very different from the actual function, resulting in sampling infeasible regions. For this reason we prefer to reserve this feature in passive learning setup, where the quantity of the data collected is enough to fit a reliable surrogate model.

3.1.2 Test functions

In order to verify the proper functionality of the algorithm during the development we used some popular unconstrained test function for optimization. [1] These functions are:

Polynomial function
Global optimum: $f(0.769) = -0.618$ with $x \in [0, 1]$
Characteristic: simple and easy to visualize

$$f(x) = \frac{1}{2}\left(\frac{1}{10}(5x-1)^4 - \frac{2}{5}(5x-1)^3 + \frac{1}{2}(5x-1)\right) \tag{3.1}$$

Ackley function
Global optimum: $f(\mathbf{0}) = 0$

[1]A collection of test functions for optimization: `https://en.wikipedia.org/wiki/Test_functions_for_optimization`

Characteristic: many local optima

$$f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)\right) + e + 20 \tag{3.2}$$

McCormick function
Global optimum: $f(-0.547, -1.574) = -1.913$
Characteristic: optimum in a plateau

$$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1 \tag{3.3}$$

Rosenbrock function
Global optimum: $f(1, ..., 1) = 0$
Characteristic: high Lipschitz constant

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \tag{3.4}$$

3.2 Surrogate model: Neural Network regressor

In single objective optimization the objective function is defined as $f : \{\mathbb{R}, \mathbb{Z}\}^N \mapsto \mathbb{R}$ where N is the number of input variables.
In our setup we use a neural network regressor as surrogate model meaning that it can learn any function in the form $f : \mathbb{R}^N \mapsto \mathbb{R}$ as proved by the universal function approximation theorem [16].
The theorem applies to networks that have at least 1 hidden layer, nonlinear activations and enough neurons to fit the data. For this reason our NN is a multi layer perceptron (MLP) with ReLU activations that is also a model perfectly supported by EML.

Neural networks present an advantage respect to kernel based surrogate models because they do not require prior knowledge of the problem, while methods like Gaussian Process requires a proper kernel selection.

3.2.1 Neural probabilistic models

Methods like Bayesian Optimization rely on stochastic surrogate models. This means that the prediction of the model is the expected value and the confidence of the prediction.
This is inherited from the usage of the multivariate Gaussian distribution and allows the algorithm to balance between exploitation (where the expected value is lower) and exploration (where the confidence interval is higher).

In order to reproduce such feature in our setup we rely on neural probabilistic models.
While in classification tasks we get as output logits the probability scores of each class, in a regression task we obtain only the raw predictions with no clue about the confidence of the model.

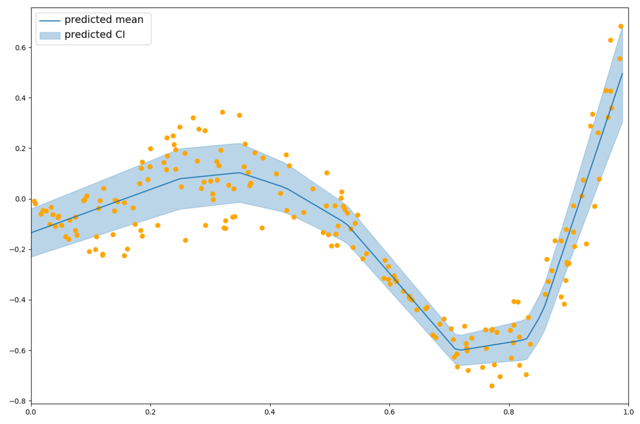


Figure 3.1: The plot represent a probabilistic model fitted on the orange datapoints. The blue curve is the output mean ± the standard deviation.

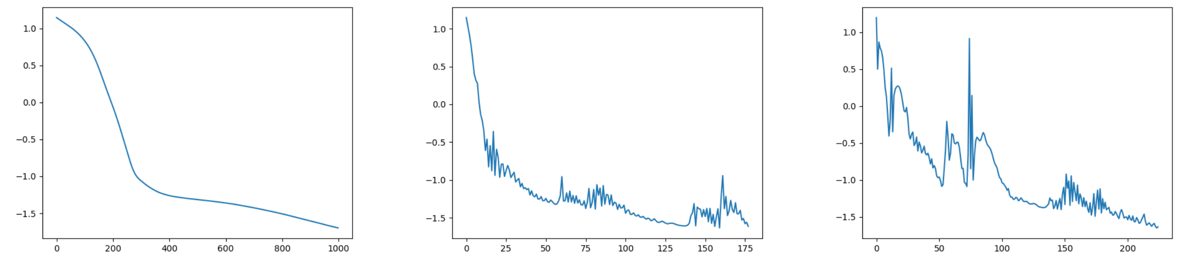
Figure 3.2

A neuro-probabilistic model is essentially a classic NN regressor trained with a different loss: instead to minimize the MSE, we try to maximize the likelihood of the data-points according to an output distribution.

The output of the NN will be the distribution parameters while the distribution that better fit the nature of the data is chosen among several ones.
For example in order to predict the probability of a given number of events occurring in a fixed interval of time we would choose the Poisson distribution. In general by assuming that the samples come from i.i.d. random variables, we use the Normal distribution, meaning that the NN outputs will be the mean and the standard deviation of the probability distribution (fig. 3.2). In order to force the standard deviation to be a positive value we apply the exponential operator to the output. This technique can be appreciated also in the field of computer vision when in object detection the width and height of the bounding boxes (which are measure, so positive values) must be predicted. Even if apply a ReLU activation function could seem straightforward, the usage of such activation in the last layer of a model cause a significant worsening of the performance because all the gradients in the negative region are lost.

Luckily Tensorflow (the Deep Learning framework that we are using) offers a package called TFProbabilistic, where this technique is easily implementable by placing a Distribution layer as the last one. The advantage is that the network output will not be just two real numbers (the mean and the stddev) but a distribution object that can be queried and used to calculate the likelihood of the labels according the predicted PDF which is used as training loss (eq. 3.5).

$$\mathcal{L} = -\log P(y|\mathcal{N}(\mu(x), \sigma(x)^2)) \tag{3.5}$$



(a) 10^{-4} (b) 10^{-3} (c) 10^{-2}

Figure 3.3: Learning rate effects on the loss

3.2.2 Surrogate model training

A very important step in the training of a NN regressor is to reduce both the input and the output of the network in the $[0,1]$ interval for numerical stability reasons.
In the contest of the optimization loop this is also useful because allows to define thresholds, coefficients and compare metrics regardless the numerical range or the objective function.
In particular, inputs and outputs are min-max scaled in the $[0,1]$ range. The fact that this transformation is linear allow to express it also in the MILP model and that comes handy when dealing with normalized integer variables.

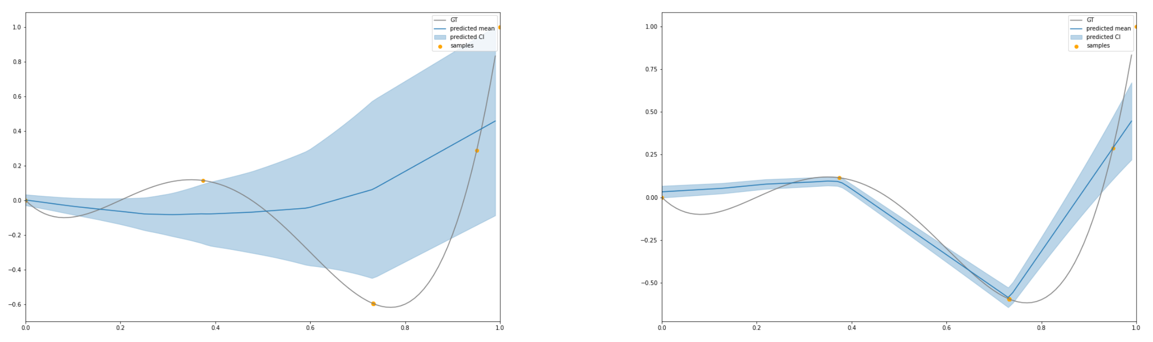
The network is trained from scratch with gradient descent instead than stochastic gradient descent because the initial dataset could be composed by a few of data-points; beside that we want to "overfit" the data so that we don't need the regularization effect of the noise introduced by the SGD.

The model consists in a MLP with 1 hidden layer 200 neurons wide. We observed that a deeper model doesn't improve significantly the performance but increase the solver time.
Training is performed with the AdamW [17, 18] optimizer, which correct the weight decay implementation present in Adam, with a constant learning rate `5e-3` and a weight decay of `1e-4` for a maximum of 1000 epochs. Since the batch size is fixed, the learning rate is the most incisive hyper-parameter: if is too low then the stop condition is never met, otherwise the training is unstable and the loss is chaotic (fig. 3.3).

We ideally want to overfit the data in order to achieve an accurate model, however we risk to incur in predictions with 0 variance that will make UCB inapplicable.

The strategy deployed consists in relying on a custom stop condition instead than the early stopping based on the validation loss: a callback will stop the training only if the average of the standard deviations calculated on the sample points is below a certain threshold.
Besides that, early stop will perform poorly when the dataset consists in an handful of points. 3.4

Another advantage of this method is that it make unnecessary the validation set which are precious data points in the context of BBO. A comparison among several test function is presented in fig 3.5.



(a) Early stop (b) Average confidence interval

Figure 3.4: Stop condition comparison. Polynomial problem

3.2.3 Model dimension and sample complexity

While a neural network is easier to tune respect to kernel methods, it still require to choose an appropriate number of parameters. This number affect directly the training performance and the MILP solver time. Depending on the number and the distribution of the data-points, a model too complex would result in an unnecessarily complicated MILP model, while if it is too simple it could underfit the data and require long time to converge.

The architecture that we are using is a MLP defined by the number of hidden layers and the number of neurons of a layer.
Let N be the number of neurons per layer, L the number of layers and D the dimensions of the input, then the number of total parameters P can be calculated as in equation 3.6.

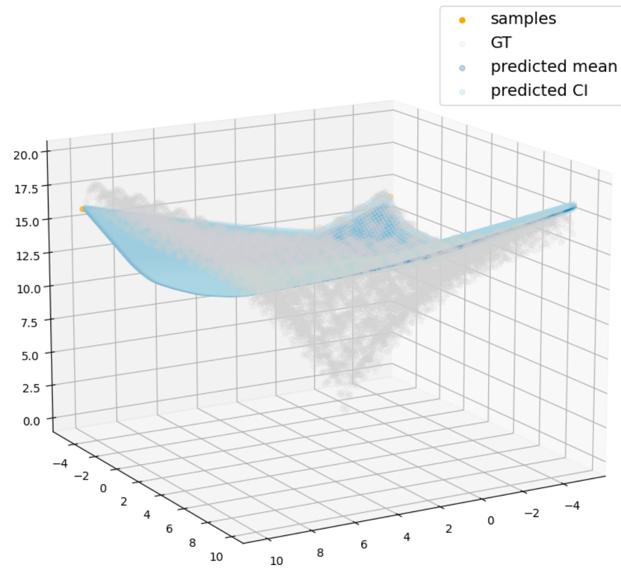
$$\begin{aligned} P &= (D+1)N + N(N+1)(L-1) + 2(N+1) \\ &= \mathcal{O}(LN^2) \end{aligned} \tag{3.6}$$

Increasing the number of neurons improve the expressiveness while increasing the number of layer improve also the ability to handle non-linearities.

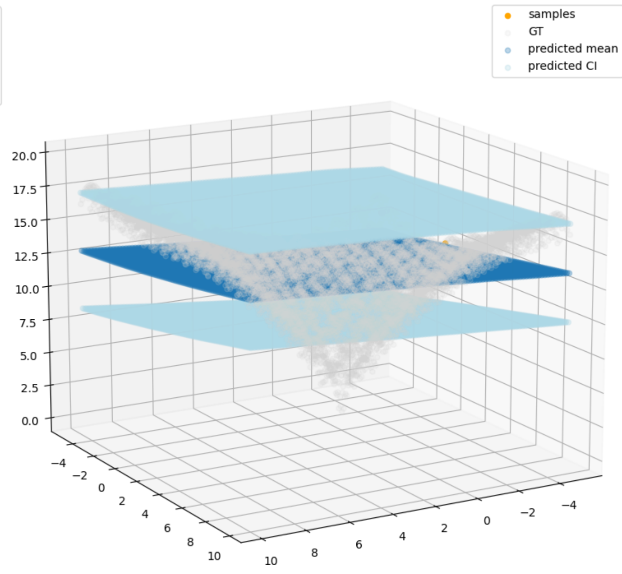
The possibility to properly tune the model capacity respect to the dataset it's interesting not only because it simplify the configuration of the search algorithm, but also because in the active learning scenario we have the chance to size the model at each iteration. We could use initially a simple model and then increase the number of parameters gradually with the amount of samples.

In Computational Learning Theory is defined the VC-dimension (VCD) as the measure of the capacity (or expressiveness) of a binary classification model. It provides an upper bounds to the maximum number of points that a model can shatter, while it is easier to compute than the Rademacher complexity. [19]

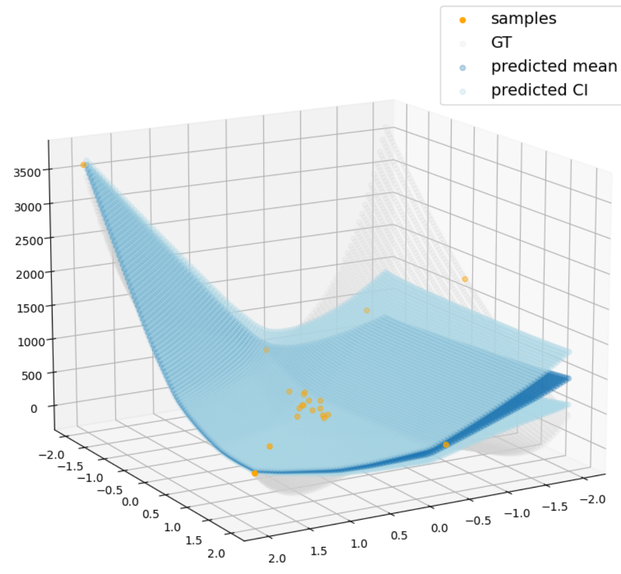
Even if the VCD is meant for classification tasks, it can be adapted to handle regression tasks using a pseudo-dimension: to find the VC dimension of a class of real-valued functions, one can find the VC dimension of the class of indicator functions that can be formed by



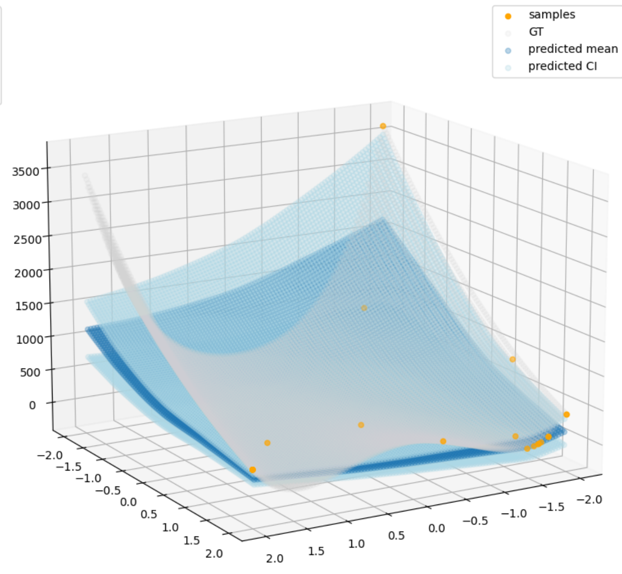
(a) Stop CI - Ackley



(b) Early stop - Ackley



(c) Stop CI - McCormick function



(d) Early stop - McCormick function

Figure 3.5: Stop condition comparison on 2D functions

thresholding that class of real-valued functions. In equation 3.7 $\mathcal{G}$ is the concept class of the regressor, $g(x)$ is the regressor output and t is a threshold.

$$Pdim(\mathcal{G}) = \text{VCD}(\{(x,t) \mapsto 1_{g(x)-t>0} : g \in \mathcal{G}\}) \tag{3.7}$$

The VCdim for a MLP with ReLU activations is equal to the number of its parameters, but for the pseudo dimension we still didn't found a practical way to calculate the value from the number of parameters and this is in fact one of the goals of the future works.

However even without a formal framework like the Learning Theory, a good indication of how the model is performing is the loss itself. In non-stochastic experiments, an high loss value is a signal for an hard to learn collection of samples.

3.3 Declarative model: MILP

At the core of the MILP model there is the EML encoding of the surrogate model.
The NN is trained with normalized data, thus the inputs are real variables bounded in the $[0, 1]$ interval. Variable scaling is realized inside the MILP model, allowing to enforce the integer type to the variables (see algorithm 2).

Once embedded the model we performs bound propagation starting from the decision variables in order to obtain tighter variable bounds. This step is crucial to the solver performance because it affect the ability of the branch and bound algorithm to prune nodes.

Algorithm 2 Surrogate model embedding

```
$input\_list \leftarrow$ empty list
$nn\_input\_list \leftarrow$ empty list
for each $var \in variables$ do
    $input \leftarrow$integer-or-real_variable$(lb = var_{lb}, ub = var_{ub})$
    $nn\_input \leftarrow$real_variable$(lb = 0, ub = 1)$
    add_constraint$(input == nn\_input * (var_{ub} - var_{lb}) + var_{lb})$
    $input\_list$.append$(input)$
    $nn\_input\_list$.append$(nn\_input)$
end for
$output_{mean} \leftarrow$real_variable()
$output_{stddev} \leftarrow$real_variable()
nn_embed$(model, nn\_input\_list, output)$
```

Domain specific constraints are then added to the model and finally is specified an objective function to minimize: in order to balance the exploration/exploitation trade-off, the output mean and standard deviation of the model are combined into an acquisition function.

The encoding of a neuro-probabilistic model with the EML is pretty trivial: we encode the NN without the distribution layer that is not supported. In this way the output of the encoded model will be the expected value and the *log* of the stddev.

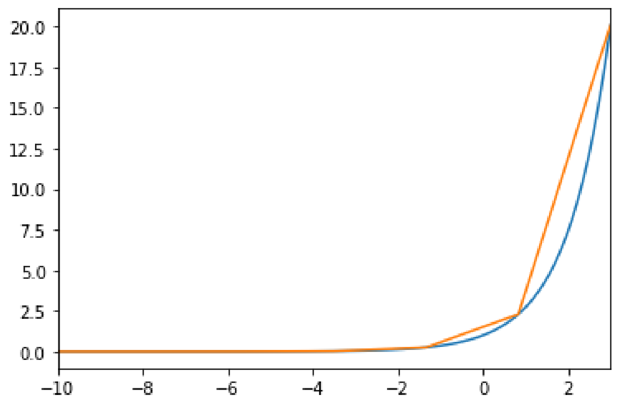


Figure 3.6: Approximation with 7 nodes. The error grows rapidly with the argument.

In order to reproduce the *exp* function in a MILP environment we resort to the SOS2 constraints to implement a PieceWise Linear function (PWL) approximation of the exponential curve.
Special Ordered Sets of type 2 (SOS2 or S2) are an ordered set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering. Special Ordered Sets of type 2 are typically used to model non-linear functions of a variable in a linear model. They are the natural extension of the concepts of Separable Programming, but when embedded in a Branch and Bound code enable truly global optima to be found, and not just local optima. [20]

Due to the linear nature of the PWL exponential, it is important to keep under control the approximation error: by increasing the number of nodes of the PWL function we can obtain a more accurate approximation at the expenses of the model complexity (fig. 3.6).

In practice the necessity to fit tightly the dataset cause the neuro-probabilistic models to have significantly high confidence intervals only at the boundaries of the domain, while this is low when interpolating between data-points.
This happens because unlike kernel based methods which are designed to increase the CI in regions far from the data, the NN loss does not decrease when increasing the CI in empty region of the space.
This characteristic is addressed in section 3.3.2 via MILP model, however the CI is still useful in situation where many points are close together or the problem has a stochastic nature.

3.3.1 Acquisition function: Upper Confidence Bound

Among the several acquisition functions used in Bayesian Optimization and described in section 2.2.2, we choose to borrow the Upper Confidence Bound (UCB) for its simplicity and linearity.
We tried also more complex acquisition functions like the Expected Improvement that relies on Gaussian CDF and PDF. Despite both of them can be represented in MILP as PWL approximations, the EI results in a quadratic non-convex formulation which thus cannot be

optimized.

The MILP model is configured to maximize an objective which is a linear combination of mean and standard deviation (equation 3.8).

$$\begin{aligned} \max \quad & -\mu + \lambda\sigma && (3.8) \\ \text{s.t.} \quad & g_j(x) \qquad \forall j \in J && (3.9) \\ & \mu, \sigma = h(x) && (3.10) \\ & x \in \{\mathbb{R}, \mathbb{Z}\}^D && (3.11) \end{aligned}$$

Where D is the domain dimensionality, g_j are problem specific constraints and h is the surrogate model learned from the samples. The mean μ is negated because the optimization loop is designed to minimize a target function (maximization can be achieved by negating the objective function).

The value of λ weights solutions with a lower expected value (exploration) or solution with an higher confidence interval (exploitation). Tuning of this value will be tackled in the following chapters, while for initial tests on simple function $\lambda = 1$ worked reasonably well.

3.3.2 Confidence interval in unexplored regions

Unlike the Gaussian Process where kernels like RBF increase the variance exponentially w.r.t. the distance to the samples, a NN has no reason to output high confidence intervals in regions that aren't covered by the dataset. This is problematic for the acquisition function because prevent the model from sampling in unexplored region.

In order to address this issue we tried two separate strategy:

1. Augment the dataset with uniformly random sampled points. Added datapoints will have a lower sample weight respect to the true samples.
2. Embed the uncertainty directly in the UCB expression by including the distance from the closest point of the dataset.

The first method works reasonably well but when dealing with high-dimensional problem, the sampling requires more and more points and the training becomes time demanding. Besides that it doesn't guarantee that the uncertainty increases monotonically with the sample distance.
Another issue with this method is the introduction of other two search parameters which are the number of points and the sample weights to assign to the true points.
The noise injected in this way interacts directly with the stop condition on the C.I., making the algorithm harder to tune.

In our setup we experimented by adding an amount of uniformly sampled random points 30 times the size of the dataset. These points are randomly sampled from the feasible space of the function and the objective value is sampled from a random uniform variable. In order to balance the added noise, each true data point has a sample weight of 60.

The second approach consists in embedding the uncertain (in unexplored region) directly in the MILP problem. The idea is to adapt the UCB such that it takes into account the distance between the proposed solution and the closest point of the dataset (equation 3.12).

$$\alpha(x) = \mu(x) + \lambda\left(\sigma(x) + dist(x, \mathcal{D})\right) \tag{3.12}$$

Where $\mathcal{D}$ is the dataset and *dist* represent the L^1 distance between the input and the closest datapoint.
The distance is implemented with a MILP real variable.

For each sample, an inequality constraint is added to the model stating that *dist* must be less or equal than the distance between the input and the sample, so since we are maximizing *dist*, only the stricter bound will be considered thus the point closest to the input (equation 3.13).

$$dist \leq \|s - x\|_1 \quad \forall s \in \mathcal{D} \tag{3.13}$$

We choose the L^1 norm respect to other metrics [2] because it is easy to implement and the only challenge is represented by the absolute value.

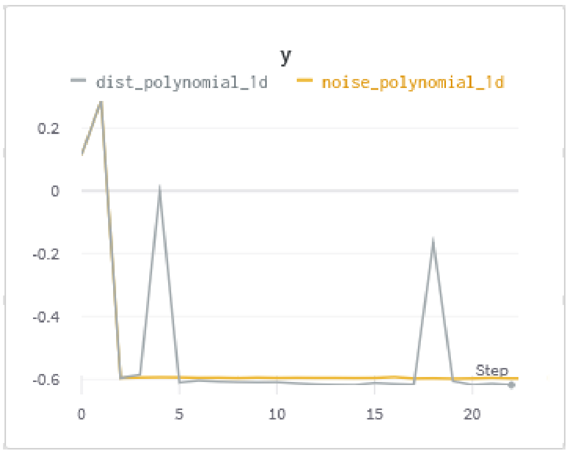
Although it is possible to implement the euclidean distance (L^2 norm) with Quadratic Programming, such distance can be only minimized while in our case we are interested in maximizing it. In QP we can minimize a concave problem or maximize a convex one and our setup results in a maximization of a non convex problem, making the model unfeasible. [21]

The absolute value can be encoded using: a solver primitive like the `abs` function provided by IBM CPLEX, an expression with a binary variable (eq. 3.14), SOS1 or SOS2 constraints.

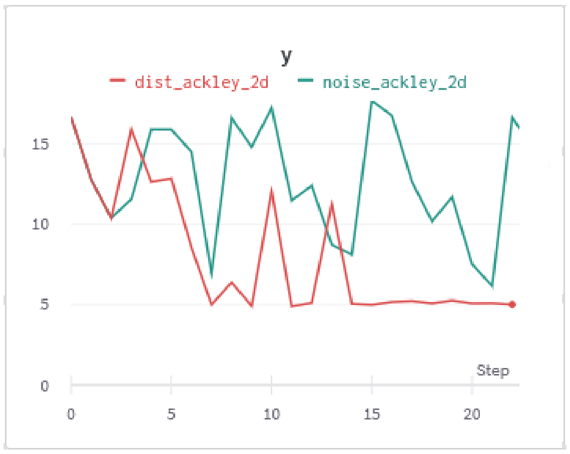
By performing a comparison between these methods we found out that the binary variable method is consistently the fastest.
Independently from the method, in order to handle the non linearity of the absolute value it adds to the model a number of boolean variable equals to the number of datapoints times the number of dimensions ($\#\mathcal{D} \cdot D$). Integer variables and in particular boolean variables make the solution exponentially slower due to the branch and bound algorithm adopted by MIP solvers.

$$\begin{aligned} x + M \cdot b &>= abs \\ -x + M \cdot (1 - b) &>= abs \\ x &<= abs \\ -x &<= abs \\ &\quad b \in \{0, 1\} \\ &\quad M \gg x \end{aligned} \tag{3.14}$$

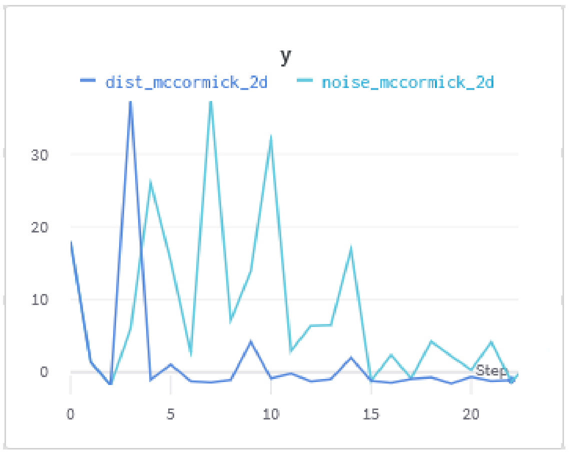
[2] L^p norms are defined as $L^p = \left(\sum_{i=1}^{d} |x_i^p|\right)^{\frac{1}{p}}$



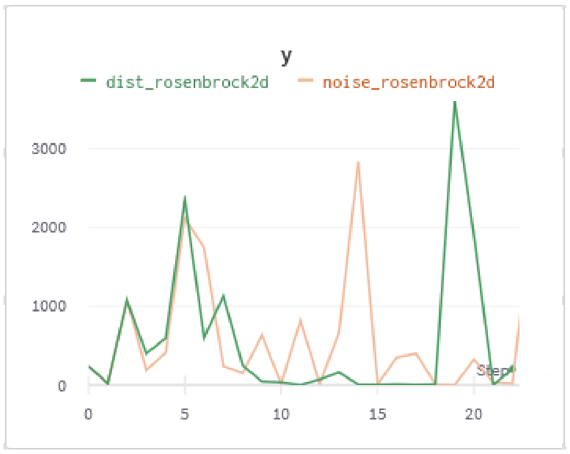
(a) Polynomial 1D



(b) Ackley



(c) McCormick



(d) Rosenbrock

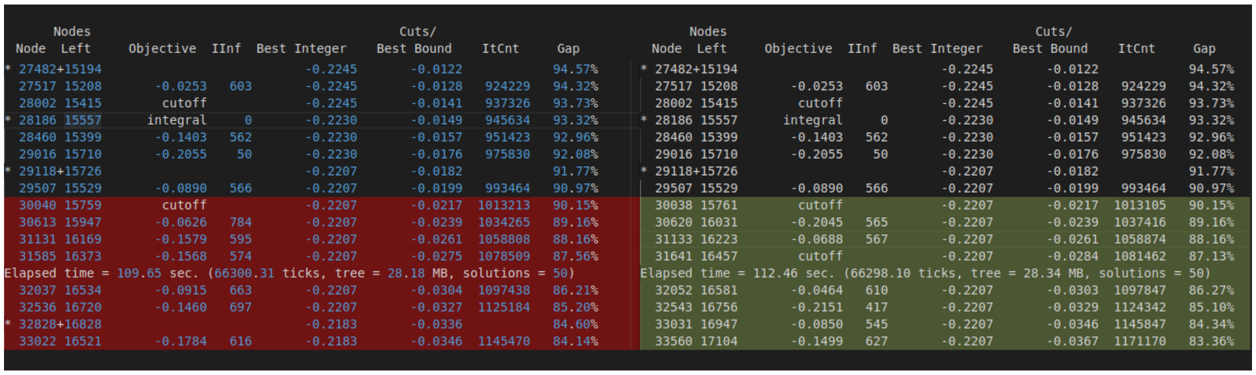
Figure 3.7: Unexplored region strategy - objective value comparison

It is worth to notice that the absolute value can be reduced to only linear constraint when minimizing it, which is not our case. [22]

Due to the curse of dimensionality [23] the distances increase in high dimension domains. In order to balance this effect the distance calculated with the L^1 norm is divided by the number of dimension (eq. 3.15). This makes the value of the distance comparable to the mean and standard deviation regardless the dimensionality of the objective function.

$$\alpha(x) = \mu(x) + \lambda \left(\sigma(x) + \frac{dist(x, \mathcal{D})}{D} \right) \tag{3.15}$$

A comparison between the noise augmentation and the embedded distance strategies is presented in fig. 3.7. The plots shows the value of the objective function during the search, despite the low dimensional problem the distance method already performs better.



```
      Nodes                                         Cuts/
 Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap
* 27482+15194                         -0.2245       -0.0122              94.57%
  27517 15208       -0.0253   603       -0.2245       -0.0128   924229    94.32%
  28002 15415        cutoff             -0.2245       -0.0141   937326    93.73%
* 28186 15557      integral     0       -0.2230       -0.0149   945634    93.32%
  28460 15399       -0.1403   562       -0.2230       -0.0157   951423    92.96%
  29016 15710       -0.2055    50       -0.2230       -0.0176   975830    92.08%
* 29118+15726                           -0.2207       -0.0182              91.77%
  29507 15529       -0.0890   566       -0.2207       -0.0199   993464    90.97%
  30040 15759        cutoff             -0.2207       -0.0217  1013213    90.15%
  30613 15947       -0.0626   784       -0.2207       -0.0239  1034265    89.16%
  31131 16169       -0.1579   595       -0.2207       -0.0261  1058808    88.16%
  31585 16373       -0.1568   574       -0.2207       -0.0275  1078509    87.56%
Elapsed time = 109.65 sec. (66300.31 ticks, tree = 28.18 MB, solutions = 50)
  32037 16534       -0.0915   663       -0.2207       -0.0304  1097438    86.21%
  32536 16720       -0.1460   697       -0.2207       -0.0327  1125184    85.20%
* 32828+16828                           -0.2183       -0.0336              84.60%
  33022 16521       -0.1784   616       -0.2183       -0.0346  1145470    84.14%
```

```
      Nodes                                         Cuts/
 Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap
* 27482+15194                           -0.2245       -0.0122              94.57%
  27517 15208       -0.0253   603       -0.2245       -0.0128   924229    94.32%
  28002 15415        cutoff             -0.2245       -0.0141   937326    93.73%
* 28186 15557      integral     0       -0.2230       -0.0149   945634    93.32%
  28460 15399       -0.1403   562       -0.2230       -0.0157   951423    92.96%
  29016 15710       -0.2055    50       -0.2230       -0.0176   975830    92.08%
* 29118+15726                           -0.2207       -0.0182              91.77%
  29507 15529       -0.0890   566       -0.2207       -0.0199   993464    90.97%
  30038 15761        cutoff             -0.2207       -0.0217  1013105    90.15%
  30620 16031       -0.2045   565       -0.2207       -0.0239  1037416    89.16%
  31133 16223       -0.0688   567       -0.2207       -0.0261  1058874    88.16%
  31641 16457        cutoff             -0.2207       -0.0284  1081462    87.13%
Elapsed time = 112.46 sec. (66298.10 ticks, tree = 28.34 MB, solutions = 50)
  32052 16581       -0.0464   610       -0.2207       -0.0303  1097847    86.27%
  32543 16756       -0.2151   417       -0.2207       -0.0329  1124342    85.10%
  33031 16947       -0.0850   545       -0.2207       -0.0346  1145847    84.34%
  33560 17104       -0.1499   627       -0.2207       -0.0367  1171170    83.36%
```

Figure 3.8: Comparison between the solver logs of two identical CPLEX model, evidencing the beginning of the divergence.

3.3.3 Improving the distance encoding

Once implemented the distance method, the solver became the bottleneck of the algorithm due to the amount of integer variables. In this section we present three strategies that we designed to mitigate the problem.

Before dive into the details of the strategies employed, it is necessary to make a note about the non-deterministic behaviour of the CPLEX solver.

In order to achieve the determinism and so the repeatably of the experiments, we fix the seed and limit the number of thread to one (due to the non associativity of the floating point operation) [24]. Also CPLEX allows to set a deterministic time limit expressed in ticks in the place of the classic one represented in seconds. The "tick" represent the most granular unit of execution during the branch and bound algorithm. Using a maximum number of ticks as limit makes the execution result invariant to the CPU load.

However, even with this precautions we were unable to exactly reproduce the experiments, probably due to a bug of the current CPLEX version (fig. 3.8) [3].

Triangular inequality

Since we are dealing with a metric measure, we try to exploit the triangular inequality: let x and y be two measures, then $\|x + y\| \leq \|x\| + \|y\|$.

In order to implement it in the model, for each pair of datapoints we compute the L_1 distance offline and then we add the constraints to the model (alg. 3).
The idea is to reduce the search space by adding more constraint to the problem, however the MILP solver improvements are driven by better bounds, which the triangular inequality doesn't improve.

Binary configurations

In this method we exploit the nature of the absolute value to reduce only the domain of the binary variables.

[3]https://stackoverflow.com/questions/71188150/achieve-true-determinism-in-cplex

Algorithm 3 Triangular inequality

```
for each $x \in points$ do
    for each $y \in points$ do
        $dist_{xy} \leftarrow dist_{L-1}(x, y)$                         ▷ real value
        $dist_x \leftarrow dist_{L-1}(input, x)$                        ▷ MILP variable
        $dist_y \leftarrow dist_{L-1}(input, y)$                        ▷ MILP variable
        $add\_constraint(dist_x \leq dist_y + dist_{xy})$    ▷ add constraint to the problem
    end for
end for
```

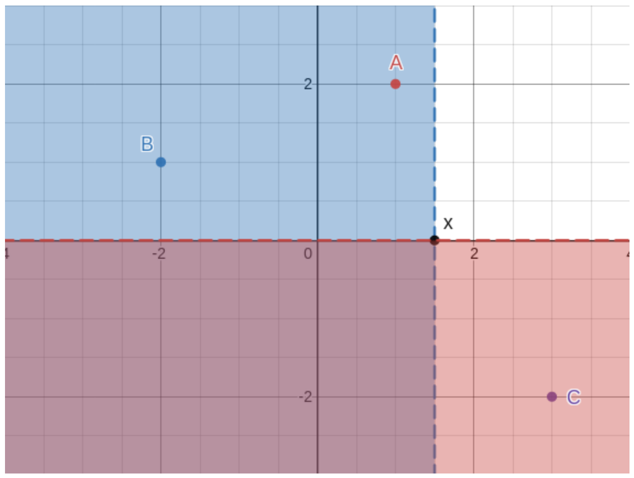


Figure 3.9: Sorting the sample points for each dimension separately, the `abs` binary configuration will be [B:1, A:1, C:0] for the x-axis and [C:1, B:0, A:0] for the y-axis.

We notice that for a given solution, each dimension of the hyperspace is split in two parts: all samples in one half will have the respectively absolute value binary variable asserted and the other half don't (fig. 3.9).

Thus we sort the datapoints separately for each dimensions and then enforce the constraints in equation 3.16, where D is the problem dimensionality, p_{ik} is the value of the i-th datapoint for the feature k and b_{ik} is the respective binary variable associated to the absolute value. The result is that the only configurations of the respective binary variables admissible are in the form `[000...00], [100...00], [110...00],...,[111...11]` reducing the number of configurations from 2^n to just n.
Unfortunately, also this method don't improve the solver time.

$$\begin{aligned} &\forall k \in D \\ &\forall i, j \quad p_{ik} \leq p_{jk} \\ &b_{jk} = 1 \implies b_{ik} = 1 \\ &b_{ik} = 0 \implies b_{jk} = 0 \end{aligned} \tag{3.16}$$

Incremental dataset

Finally we implement the incremental strategy: since we rely on the distance between the input and only the closest point of the dataset, we designed a procedure that solve a model with only a subset of the datapoints and then incrementally adds one sample at time until the closest point to the solution found is already included in the subset (alg. 4).

Algorithm 4 Incremental dataset

```
$selection \leftarrow$ empty_list
repeat
    $solution \leftarrow$solve($selection$)                                    ▷ sub problem solution
    $closest\_selected \leftarrow$argmin(dist($solution, selection$))
    $closest\_total \leftarrow$argmin(dist($solution, samples$))
    $selection$.append($closest\_total$)
until dist($closest\_total, solution$) $\geq$ dist($closest\_selected, solution$)
```

This method revealed to be the one that consistently improves the solver time.
In the experiment chapter we compare the behavior of the standard distance embedding and the incremental.

3.3.4 UCB and Lipschitz constant

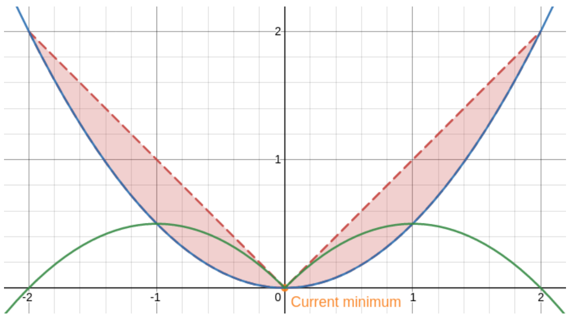
In previous experiments we used the UCB acquisition function with $\lambda = 1$. This gives the same weight to exploitation (the μ value) and exploration ($\sigma + dist$).

In practice the ideal λ value depends on the slope of the function: for a function that tends to change rapidly it is worth exploring regions also if the contribution of the estimated objective value to the acquisition function is significantly smaller than the distance from the dataset, so could be better to use an higher value for λ.
In fig. 3.10 we represent an example surrogate model with expected value $y = \frac{1}{2}x^2$ for different λ values, showing how it impacts the acquisition function.

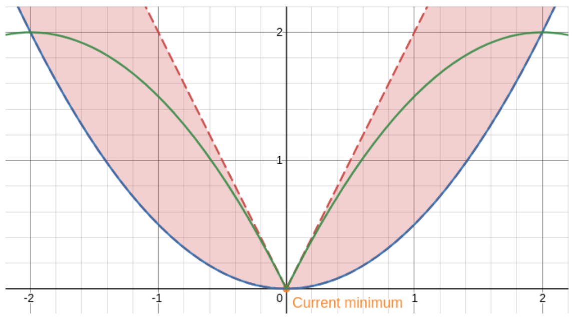
In order to provide a formal method to balance exploration and exploitation depending on the objective function, we rely on the concept of Lipschitz constant.

A Lipschitz continuous function is limited in how fast it can change: there exists a real number such that, for every pair of points on the graph of this function, the absolute value of the slope of the line connecting them is not greater than this real number; the smallest such bound is called the Lipschitz constant of the function (eq. 3.17) [25]. It can be interpreted as a generalization of the derivative since it is not bounded to be computed between points in close proximity, instead it is a global measurement.

$$\frac{d_y(f(x_1), f(x_2))}{d_x(x_1, x_2)} \leq K \quad \forall x_1, x_2 \quad x_1 \neq x_2 \tag{3.17}$$



(a) $\lambda = 1$



(b) $\lambda = 2$

Figure 3.10: Blue curve: the surrogate model prediction. Green curve: UCB acquisition function. Red space: region where the distance contribute is higher than the surrogate prediction.

At each iteration, we compute the Lipschitz constant by considering each pair of sample points collected and then assign the maximum K value to λ (eq. 3.18).

$$\lambda = max\left(\frac{|f(x_1) - f(x_2)| \cdot D}{\|x_1 - x_2\|_1}\right) \quad \forall x_1, x_2 \in \mathcal{D} \tag{3.18}$$

We notice also that in some cases could be better to use the 95th percentile in the place of the max because a cluster of very close points could result in an exceeding high Lipschitz constant.

3.4 Initial points sampling

The selection of the first points must respect the problem constraints and should be pseudo-random at the same time. In order to accomplish this we evaluate several strategies that are listed below.

1. Random sampling rejecting infeasible solutions

This is the simplest method. It consists in sampling uniformly random points and reject the infeasibilities. It is the easiest method to implement but may be impractical for problem with little feasible regions. One solution could be to sample according LHS but the number of points scales exponentially with the dimensions of the problem.

2. Solve a MILP instance with problem constraints

In order to guarantee the feasibility and the randomness of the solutions we setup a MILP model with the problem constraints and set as objective the linear combination of the inputs (with random weights).

The issue with this method is that due to the linearity of the objective function all the points are sampled in the boundaries of the feasible region.

This is undesirable for two reasons: first it generate a bad initial configuration because the objective values are likely to have very different values in the boundaries of the feasible region respect to the inside, resulting in surrogate model with an inaccurate global trend.
Second, when the algorithm explore regions far from the known points, it will choose points that still lie on the boundary of the feasible region. This is particularly problematic when dealing with high-dimensional problems because the number of "farthest points" lying in the boundaries scales up exponentially with the number of dimensions, meaning that there will be no sampling inside the feasible region until the $2^{D\ th}$ step.

3. Random quadratic objective function

An alternative to the linear objective function can be the use of a quadratic one: several MILP solvers can handle also Quadratic Programming problems. The advantage of quadratic objective is that the minimum can fall inside the feasible region and not necessarily on the boundaries. However this method is still vulnerable to little feasible regions: in that case points will still be sampled on the boundaries.

4. Interpolate boundaries points

This method works only for convex constraints: it consists in sampling half of the required initial points with the linear objective function resulting in boundaries points and then perform random interpolation between them until the number of required initial points is reached.

The convexity of the constraints guarantee that the interpolation between any two points is inside the feasible region.

5. Solve a restricted problem

While the above method is simple and elegant, it doesn't work for integer variables and thus for combinatorial constraints.

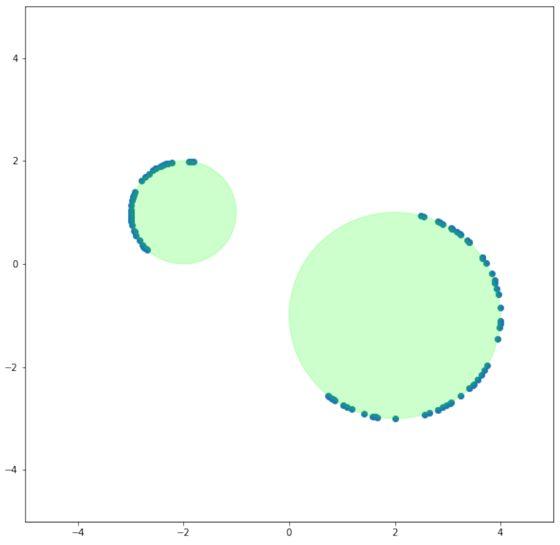
A way around the problem is to solve a restricted version of the problem: let the problem constraint be in the form $g(x) \gtreqless N$ where x is the array of input variables and N is the known term. We can multiply N by a random factor $\alpha \in [0, 1]$ if the sense of the sign is $\leq$ and $\alpha \in [1, 2]$ if the sense is $\geq$ in order to obtain a restricted version of the problem.

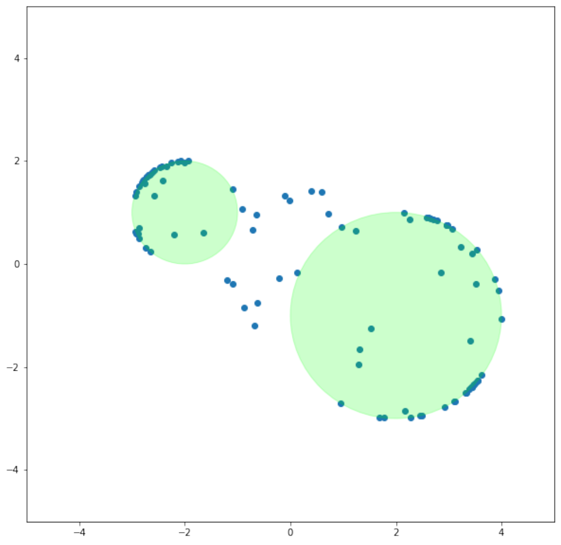
The drawback of this method is the risk to obtain many infeasible problems before getting the required number of initial points. A solution could be to shrink the interval of α around 1 as more infesabilities are collected.

About non convex constraints

In MILP non convex constraints can be easily implemented by relying on the integer variables.
Despite the EML optimization loop supports them, initialization methods explored so far don't. In figure 3.11 we show how the above initialization methods perform poorly on non convex constraints. In the figure the feasible region is defined as follows in equation 3.19.

$$\begin{aligned} (x_1 - 2)^2 + (x_2 + 1)^2 - b \cdot M &\leq 4 \\ (x_1 + 2)^2 + (x_2 - 1)^2 - (1 - b) \cdot M &\leq 1 \\ b &\in \{0, 1\} \\ M &\gg x_1, x_2 \end{aligned} \tag{3.19}$$


(a) Linear objective


(b) Linear objective + interpolation

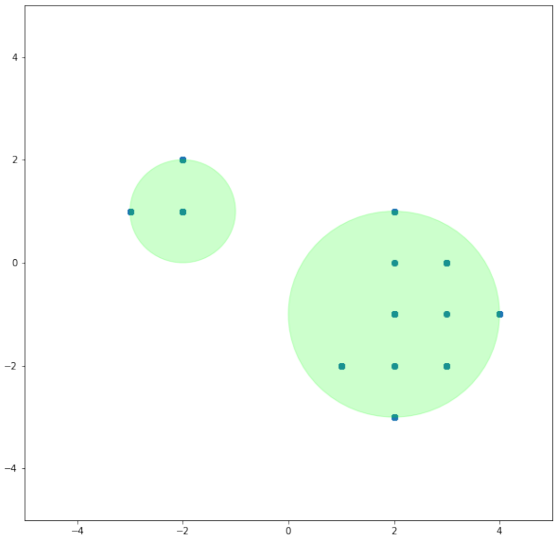

(c) Restriced integer problem

Figure 3.11: Initialization strategies comparison: sampling of 100 datapoints

Chapter 4

Experiments

4.1 Improvements

We present a couple of strategies that we analyzed that are designed to improve the quality of the solutions and the performance.

4.1.1 Lambda policy

In this heuristic we initially assign the λ value to the Lipschitz constant calculated from the samples, then at each iteration the value of λ decrease with a parabolic trend 4.1.

$$\lambda = K\left(1 - \frac{iteration}{max\_iterations}\right)^2 \tag{4.1}$$

The rationale is that we want to enforce the model to exploit more in the final steps of the search because we noted that even with a low λ value the optimization still explore due to the slightly different approximation of the NN.

4.1.2 Large Neighborhood Search

Large Neighborhood Search (LNS) is an approach that try to combine exact optimization approaches (MILP in our case) with heuristic approaches like local search strategies and it is typically used when the instances to solve are too complex.

LNS operates by trying to improve an incumbent solution. Improving solutions are searched in a local neighborhood and each neighborhood is explored via a complete search on a restricted problem.

In practice we choose randomly which decision variables to relax at each iteration by assigning them to the value of the optimum discovered so far. Then among the different sub-problems only the best solution is returned.

This variant needs an additional parameters which is the number of sub-problems solved at each iteration while the number of variable to fix is chosen accordingly.

In figure 4.1 we show that in the initial steps the LNS finds better solutions but in the long run the solver time becomes blocking.

4.2 Benchmarks

In this chapter we describe how the EML optimization loop performed on a known optimization problem used as benchmark, comparing the results with the related work listed in the SCBO paper[9].

We tested also the performance on a custom real world task noted as neural network quantization.

4.2.1 Ackley 10D

In this experiment the objective function is a 10-dimensional Ackley function with 2 convex constraints (eq. 4.2). The decision variables are all real and are bounded in the [-5, 10] interval.

$$\sum_{i=1}^{D} x_i \leq 0 \quad \wedge \quad \sum_{i=1}^{D} x_i^2 \leq 25 \tag{4.2}$$

In figure 4.2 are compared the minimum objective value at each iteration among the methods exposed in the SCBO paper [9].
The vanilla Bayesian Optimization performances are similar to the random strategy.

We replicate their setup which consists in 10 initial points and 200 iterations.
Unlike black box constraints methods, the EML optimization produces only feasible solutions. In this test case the second constraint assert that feasible points are inside an hyper-sphere of radius 5 centered in the origin. Note that due to the behaviour of the Ackley function, the objective value gets smaller in proximity of the origin, in fact our method starts from a lower value of the objective.

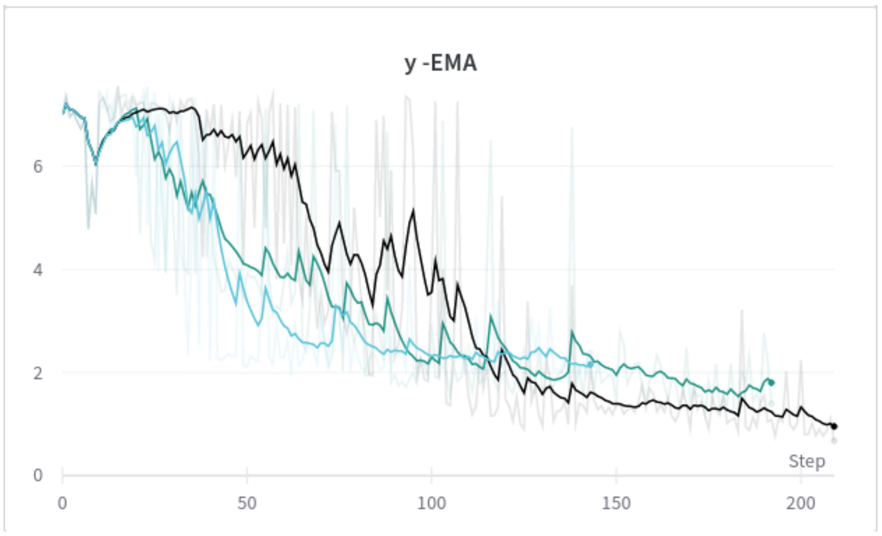
In figure 4.3 we also show that even if the CPLEX solver is not deterministic, all the methods show a similar behaviour, in particular the incremental distance is preferred over the simple distance due to the lower wall time 4.4 while the λ policy brings no improvement respect to a fixed one.

4.2.2 Neural Network quantization

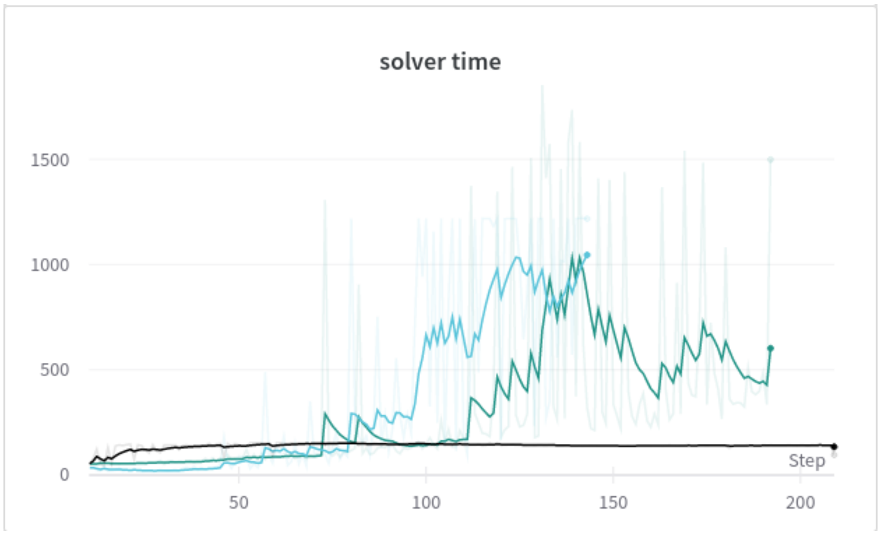
We test the EML optimization loop on the network quantization task. Similarly to the neural architecture search it consist in finding a configuration of parameter that maximize the model accuracy while respecting the constraints about the resource requirements, but is less compute demanding, allowing us to experiment with limited hardware.

Quantization background

In computer science there are several way to boost the performance of numerical methods while affecting the quality of the result as little as possible. Usually these methods lower the



(a) Objective value - EMA



(b) Solver time timeout 120s

Figure 4.1: Ackley 10D with 2 constraints - comparison between the standard implementation (black) and the LNS with 3 and 5 subproblems.

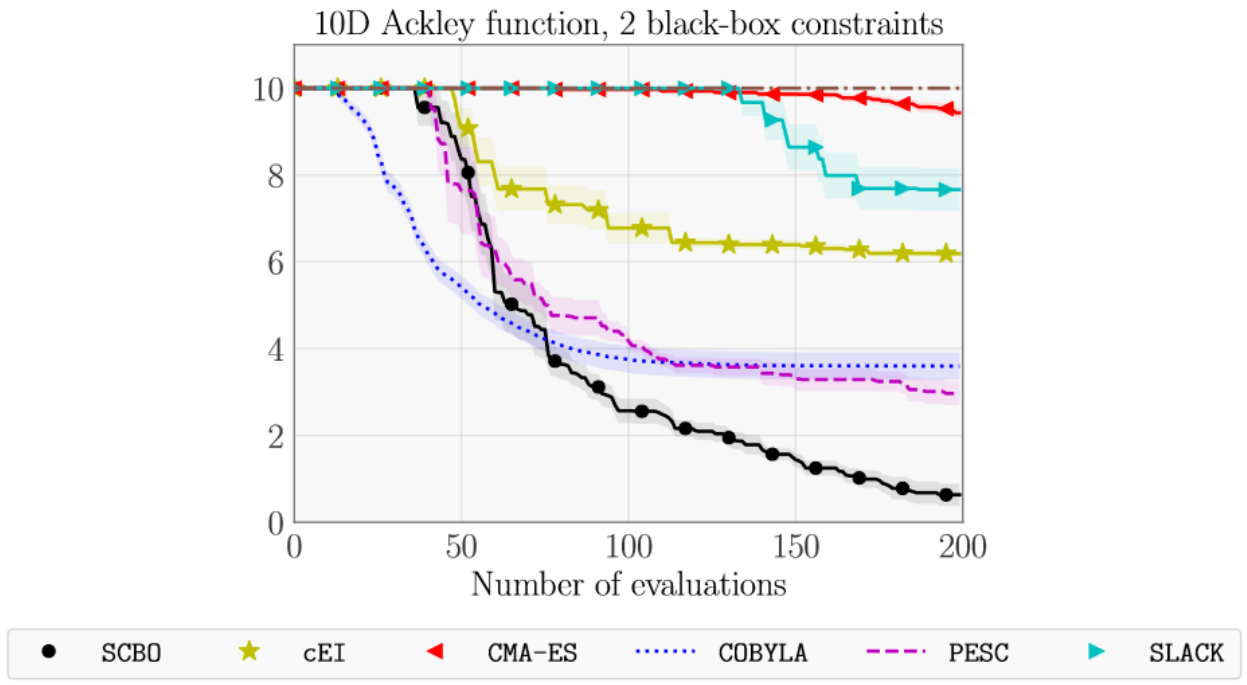


Figure 4.2: Performance comparison among SCBO methods on Ackley10D with 2 constraint [9]

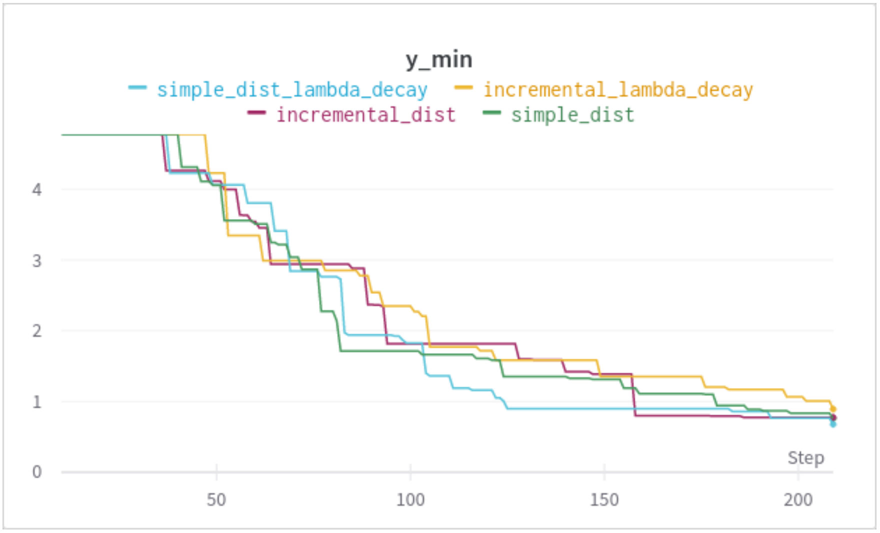
precision of the number representation in order to gain performance in terms of space and computational time. In fact storing variables using less bits not only reduce the memory footprint but also improve the processing speed because more data can be retrieved in a single bus cycle and stored in the caches.

In the deep learning field there are many different approaches which can be divided in two classes: one reduce the size of the operands (like mixed-precision training via bit-width reduction [26], quantization [27]) and the other reduce the number of operations (like network pruning [28]).

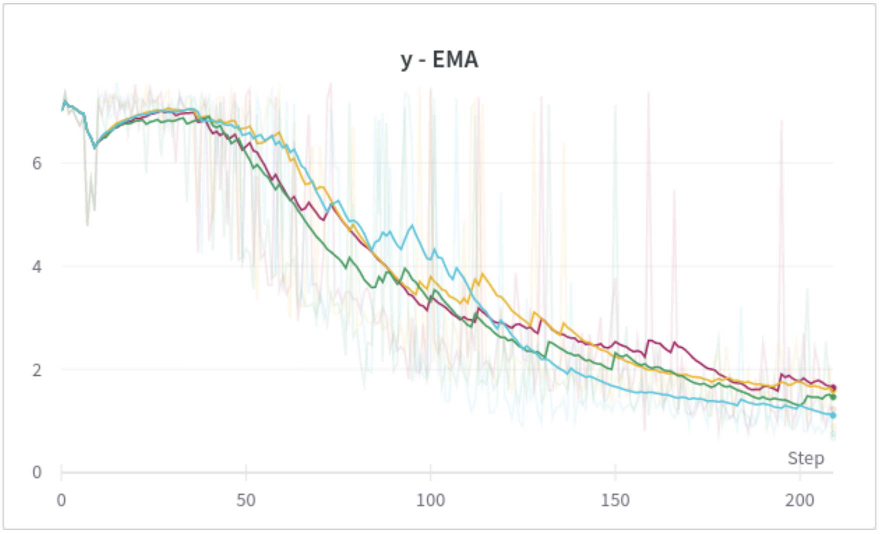
In quantization the network weights and activations that usually are represented with 32-bit floating point variables [29] are converted to the integer representation with a predetermined number of bits, where the number of different discrete values that can be represented (levels) is given by 2^{bits}. In equation 4.3 is described how the quantization Q is performed on the real variable r, where $[\alpha, \beta]$ denotes the clipping range of the real value. The process of choosing the clipping range is often referred to as calibration.

$$Q(r) = \text{Int}(r/S) - Z \qquad (4.3)$$
$$S = \frac{\beta - \alpha}{2^{bits} - 1}$$

A straightforward choice is to use the min/max of the signal for the clipping range, i.e. $\alpha = r_{min}$, and $\beta = r_{max}$. This approach is an asymmetric quantization scheme, since

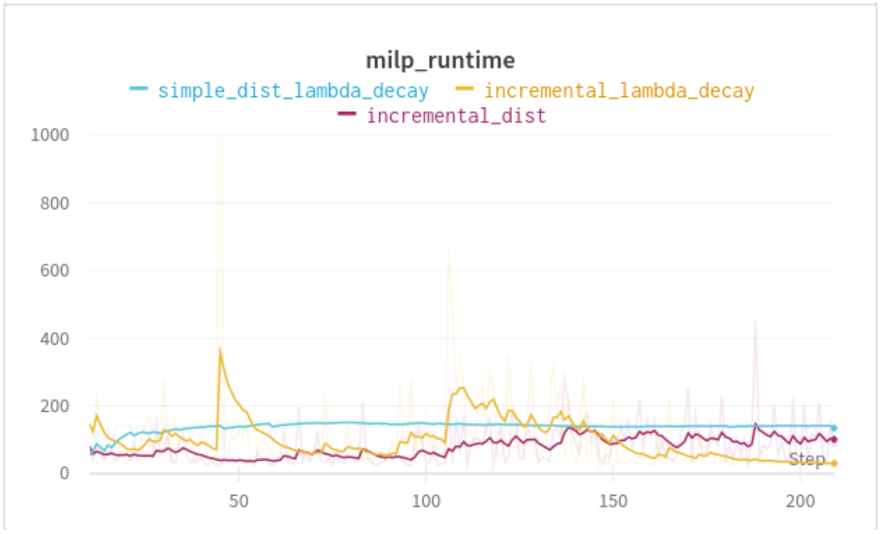


(a) Objective value - minimum

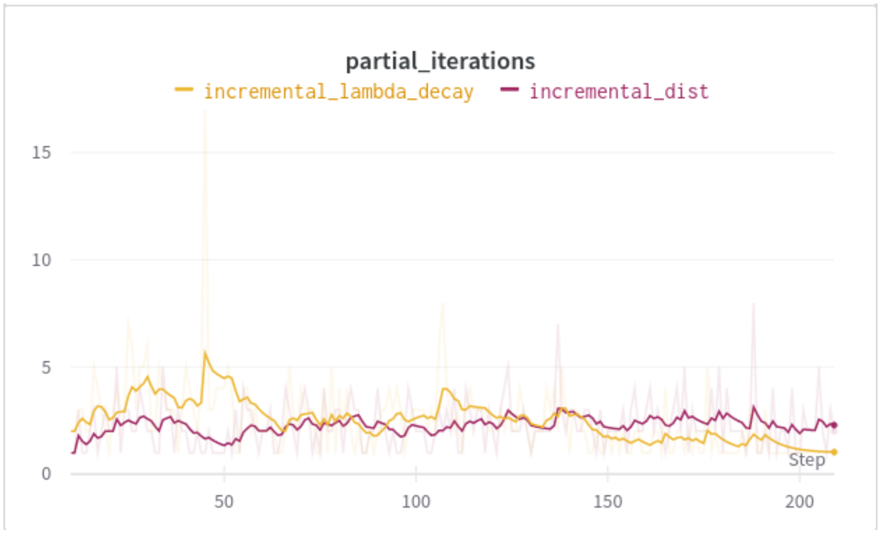


(b) Objective values - exponential moving average

Figure 4.3: Comparison between the basic and the incremental distance implementation with $\lambda = 1$ and lambda decay



(a) Solver time with timeout 120s



(b) Partial iterations with the incremental distance

Figure 4.4: Solver time comparison between standard and incremental distance encoding

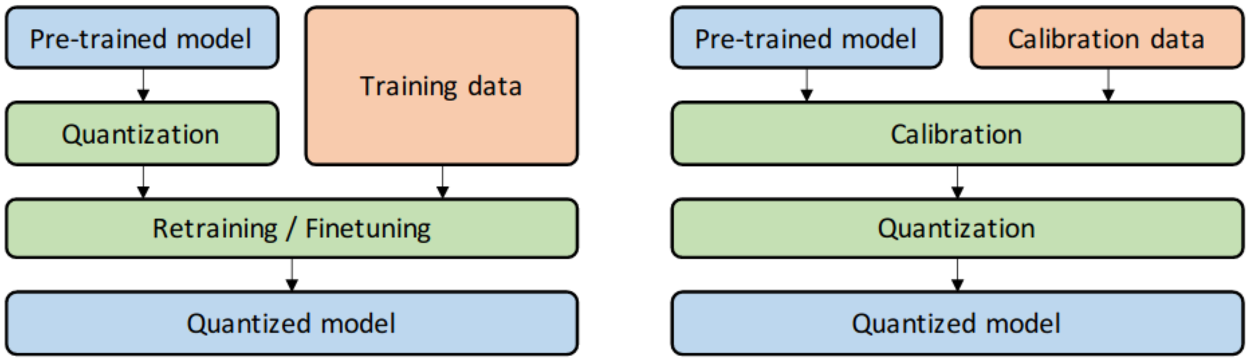


Figure 4.5: Comparison between Quantization-Aware Training (QAT, Left) and Post-Training Quantization (PTQ, Right) [27]

the clipping range is not necessarily symmetric with respect to the origin. Asymmetric quantization is preferred when the target weights or activations are imbalanced, e.g., the activation after ReLU that always has non-negative values.

In this approach, the clipping range is determined by considering all of the weights in the convolutional filters of a layer, this is noted as Layerwise Quantization.

Quantization can also be computed dynamically by clipping the range of each activation, achieving usually the highest accuracy. However, calculating the range of a signal dynamically is very expensive, and as such, practitioners most often use static quantization where the clipping range is fixed for all inputs (like our case). [27]

Finally it is often necessary to adjust the parameters in the NN after quantization. This can either be performed by retraining the model, a process that is called Quantization-Aware Training (QAT), or done without re-training, a process that is often referred to as Post-Training Quantization (PTQ) (fig. 4.5). In QAT, a pre-trained model is quantized and then finetuned using training data to adjust parameters and recover accuracy degradation. In PTQ, a pre-trained model is calibrated using calibration data (e.g., a small subset of training data) to compute the clipping ranges and the scaling factors. Then, the model is quantized based on the calibration result.

It is important to note that in QAT the usual forward and backward pass are performed on the quantized model in floating point, but the model parameters are quantized after each gradient update.

Performing the backward pass with floating point is important, as accumulating the gradients in quantized precision can result in zero-gradient or gradients that have high error. An important subtlety in backpropagation is how the the non-differentiable quantization operator is treated. Without any approximation, the gradient of this operator is zero almost everywhere, since the rounding operations.

A popular approach to address this is to approximate the gradient of this operator by the so-called Straight Through Estimator (STE). STE essentially ignores the rounding operation and approximates it with an identity function. [27]

Experiment setup

Our setup consists in performing QAT on a ResNet18 [30] convolutional network pretrained on the dataset CIFAR10 [31] for the classification task.
The model is pretrained for 50 epochs with a batch size of 128 and the Adam optimizer with a learning rate of 10^{-4}.

The quantization scheme used for the QAT is a list of 41 integer variables in the interval $[2, 8]$ that specifies how many bit must be used to quantize each layer. These values are the decision variables of the optimization. Unlike the Ackley experiment, this one is heavily based on integer decision variables.

The the objective function performs QAT with the given scheme for just 5 epochs returning the accuracy of the model evaluated on the test set. This operation requires about 5 minutes on a Nvidia V100 GPU.

Since this is just an experiment to test the behavior of the optimization loop, the problem constraint limits the number of bits to be used as described in eq. 4.4. However it is straightforward to extend the constrain to the exact memory footprint of the model, in fact for each convolutional layer, the number of parameters is given by $K_i K_o W^2$ where K_i is the number of input channels, K_o the number of output channels and W is the kernel size.

$$\sum_{i=1}^{41} x_i \leq 41 \cdot 4 \tag{4.4}$$

Results

In equation 4.5 is formalized our optimization goal. Due to the non-deterministic nature of the training on GPU, the EML optimization loop is configured to treat the objective as a stochastic function.

$$\begin{aligned} min \quad & -accuracy \\ s.t. \quad & \sum_{i=1}^{41} x_i \leq 41 \cdot 4 \\ & x \in [2, 8]^{41} \end{aligned} \tag{4.5}$$

The experiment sample 10 initial points with the restricted feasible region method and then performs 100 steps.

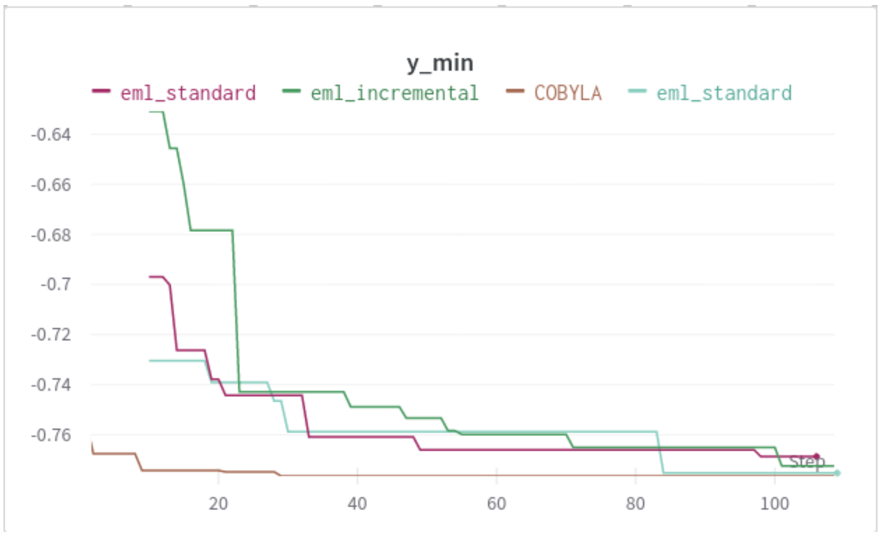
The pretrained floating point model has an accuracy of about 0.762 while a quantization scheme of 4 bits for each layer results in an accuracy of 0.755.

We encountered difficulties in finding a viable implementation of the related works because some methods doesn't support integer variables, constraints or are old libraries written in Python 2 which conflicts with Tensorflow.

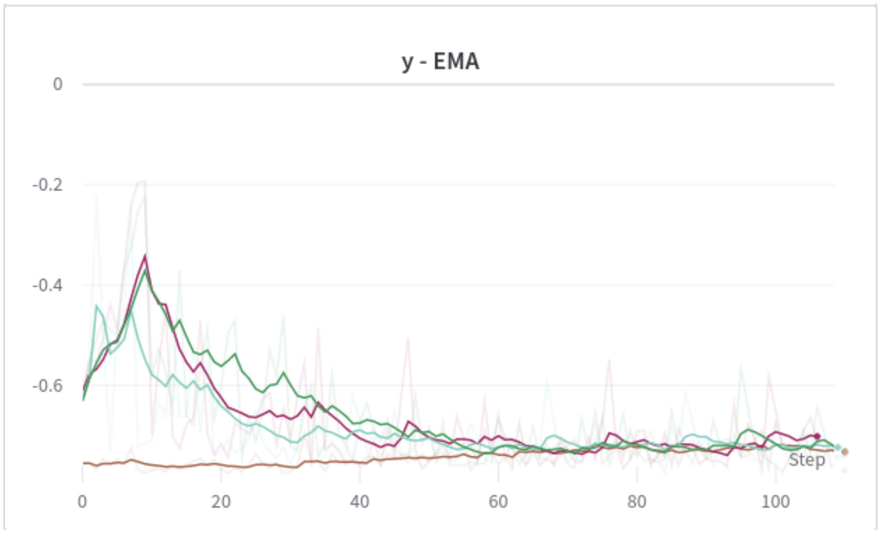
The only optimization strategy that we compared is the SciPy implementation [32] of COBYLA which requires a starting point to proceed with the optimization. We provided the four bits for each layer configuration. In figure 4.6 we compare several different EMLOpt runs against the COBYLA solver.

In figure 4.7 we observe that for problems with mainly integer variables the incremental strategy performs worse than the standard one because even a model with few samples is hard to solve, incurring in timeout at each partial iteration.

It is worth to notice that also if the solver always exceed the time limit this doesn't necessarily mean that the solution is sub-optimal, and even if it is, a good approximation it is still useful.

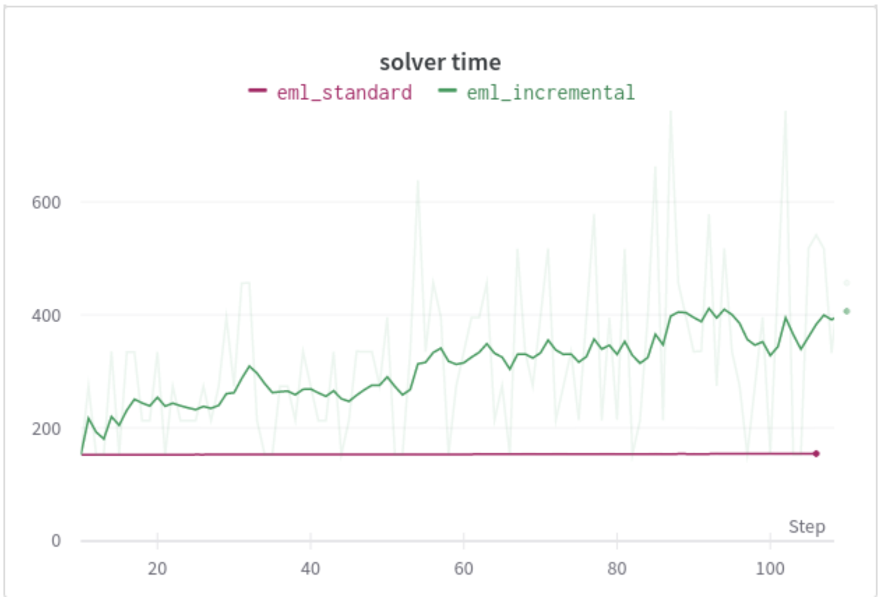


(a) Objective value - minimum

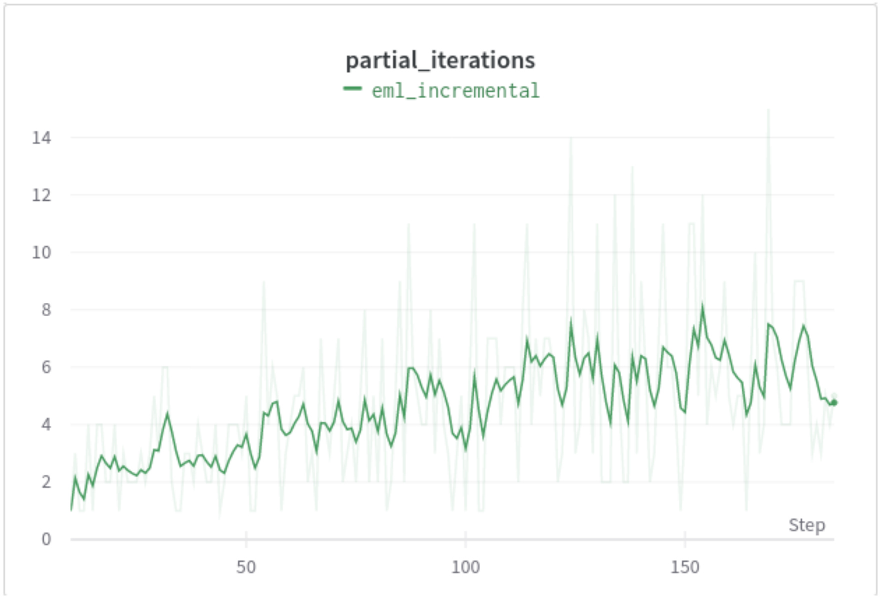


(b) Objective value - EMA

Figure 4.6: Comparison among several EML experiment with a different initialization and the COBYLA algorithm which was not randomly initialized but started with the configuration that assign 4 bits to each layer.



(a) Solver time with timeout 120s



(b) Number of partial iterations solved for incremental distance embedding

Figure 4.7: The number of partial iterations is higher w.r.t. the Ackley problem due to the low Lipschitz constant of this experiment (about 0.4).

Chapter 5

Conclusions

In this work we presented a Black Box optimization model that can handle also combinatorial constraints and is based on the Empirical Model Learning methodology.

Unlike others works that are bounded to use a specific surrogate model like the Gaussian Process, EML allows to choose the most suitable algorithm that best fit the situation, along with the prescriptive model solver.

The EML optimization loop described uses a probabilistic neural model as surrogate and mixed-integer linear programming to optimize the prescriptive model.

The solver is the main bottleneck when dealing with high dimensional problems and many samples resulting in an higher wall time respect to the other methods. However, by returning only feasible solutions this method finds use in situations where the constraint satisfaction is critical.

Among the several methods that we studied in order to improve the distance encoding the incremental procedure is consistently faster on problems with few integer decision variables while for heavily integer/combinatorial problems it is convenient to adopt the standard strategy.

Is to be kept in mind that if the solver time is too impacting, it is possible to use a configuration that ignore the distance and still obtain viable results as showed in other works [15].

However there is still a lot of work to be done. The main goal it to reduce the solver time by addressing the distance calculation in a more performing way.

In order to tighten the variables bounds and improve significantly the performance a solution can be to include the domain constraint during the bound propagation phase.

About the surrogate model, using a relatively small NN and vanilla gradient descent we obtained decent results on the tested problems. In future developments will be stressed the capacity of the model in order to derive a formal proof backed by computational learning theory which can relate the model dimension with the dataset.

Future development includes tests on combinatorial problems, like epidemic control with measures, in which the EML optimization stands out respect to the other black box constraints methods. For this kind of task will be necessary to design an initialization strategy that is able to deal with non convex constraints.

Bibliography

[1] P.M. Pardalos R. Horst and N.V. Thoai. *Introduction to Global Optimization, Second Edition*. Kluwer Academic Publishers, 2000.

[2] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*. Kluwer Academic Publishers, 2012.

[3] Jochen Görtler, Rebecca Kehlbeck, and Oliver Deussen. A visual exploration of gaussian processes. *Distill*, 2019. https://distill.pub/2019/visual-exploration-gaussian-processes.

[4] Apoorv Agnihotri and Nipun Batra. Exploring bayesian optimization. *Distill*, 2020. https://distill.pub/2020/bayesian-optimization.

[5] Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artificial intelligence*, 244, 2017-03.

[6] Indicator constraints and big-m formulation. `https://www.ibm.com/support/pages/difference-between-using-indicator-constraints-and-big-m-formulation`.

[7] Ed Klotz and Alexandra M. Newman. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, 18(1):18–32, 2013.

[8] Stéphane Alarie, Charles Audet, Aïmen E. Gheribi, Michael Kokkolaras, and Sébastien Le Digabel. Two decades of blackbox optimization applications. *EURO Journal on Computational Optimization*, 9:100011, 2021.

[9] David Eriksson and Matthias Poloczek. Scalable constrained bayesian optimization, 2021.

[10] Jacob R. Gardner, Matt J. Kusner, Zhixiang Xu, Kilian Q. Weinberger, and John P. Cunningham. Bayesian optimization with inequality constraints. page II–937–II–945, 2014.

[11] José Miguel Hernández-Lobato, Matthew W. Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions, 2014.

[12] Andrew R. Conn; Katya Scheinberg; Ph. L. Toint. On the convergence of derivative-free methods for unconstrained optimization. 1997.

[13] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772, 2016.

[14] Victor Picheny, Robert B. Gramacy, Stefan M. Wild, and Sebastien Le Digabel. Bayesian optimization under mixed constraints with a slack-variable augmented lagrangian, 2016.

[15] Theodore Papalexopoulos, Christian Tjandraatmadja, Ross Anderson, Juan Pablo Vielma, and David Belanger. Constrained discrete black-box optimization using mixed-integer programming, 2021.

[16] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[18] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[19] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[20] J.A. Tomlin. Special ordered sets and an application to gas supply operations planning. *Ketron Management Science*, 1988.

[21] Identifying a quadratically constrained program: Convexity. `https://www.ibm.com/docs/en/icos/12.10.0?topic=qcp-convexity`.

[22] Kathleen Zhou Benjamin Granger, Marta Yu. Optimization with absolute values. `https://optimization.mccormick.northwestern.edu/index.php/Optimization_with_absolute_values`, May 25, 2014.

[23] R. Bellman, R.E. Bellman, and Karreman Mathematics Research Collection. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961.

[24] What every computer scientist should know about floating-point arithmetic. `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`.

[25] M. O'Searcoid. *Metric Spaces*. Springer Undergraduate Mathematics Series. Springer London, 2006.

[26] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

[27] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.

[28] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

[29] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[31] Cifar10 - canadian institute for advanced research. `https://www.cs.toronto.edu/~kriz/cifar.htm`.

[32] Scipy optimize - cobyla. `https://docs.scipy.org/doc/scipy/reference/optimize.minimize-cobyla.html`.

[33] Weights and biases. `https://wandb.ai/site`.

Appendix A

Example problem definition

```python
from emlopt.problem import build_problem
from emlopt.search_loop import SearchLoop

def fun(x):                          # 2D McCormick function
    x1,x2 = x[0],x[1]
    return math.sin(x1+x2) + math.pow(x1-x2,2)-1.5*x1+2.5*x2+1

def cst_disk(cplex, xvars):          # domain variables inside a half-disk
    x1,x2 = xvars
    r = 0.5
    return [
        [x1*x1 + x2*x2 <= r*r, "disk_constraint"],
        [x1 <= x2, "inequality_constraint"]
    ]

problem = build_problem(
                "Problem name",           # problem name for debug
                fun,                      # objective function
                ["real", "real"],         # variables type
                [[-3, 3], [-2, 2]],       # variables bounds
                cst_disk,                 # constraint callback
                stocasthic=False)         # stocasthic flag

search = SearchLoop(problem, CONFIG)
search.run()
```

Listing A.1: Python code for problem definition

Appendix B

Parameter list

Table B.1: Search parameters

Name	**Type**	**Description**	**Default**
ITERATIONS	Integer	Maximum number of iterations	100
STARTING_POINTS	Integer	Number of initial points to sample	10
EPOCHS	Integer	Maximum number of epochs	999
SURROGATE_MODEL	Object	Surrogate model	StopCI
CI_THRESHOLD	Real	Minimum average confidence interval	0.05
LR	Real	Adam's learning rate	10^{-4}
WEIGHT_DECAY	Real	Adam's weight decay	10^{-4}
BATCH_SIZE	Integer	NN batch size	None
DEPTH	Integer	Number of hidden layers in the MLP	1
WIDTH	Integer	Number of neurons per hidden layers	200
MILP_MODEL	Object	Solver	IncrementalDist
LAMBDA_UCB	Real	Manages the exploration/exploitation	None
SOLVER_TIMEOUT	Integer	MILP sovler timeout in seconds	120

NOTE:

- Set BATCH_SIZE to `None` to train without mini-batches.
- Set LAMBDA_UCB to `None` to compute the value dynamically (see sections 3.3.4 and 4.1.1).

Appendix C

Experiment tracking

During the development of the algorithm was extremely useful to keep track of all the metrics generated from both the surrogate and the MILP model, along with the parameters of each experiment.
We relied on the Weights and Biases [33] service which allows to store in cloud and compare the plots in real time.

Since tracking the experiment is crucial to tune properly the algorithm, we designed the library such that it accept user defined callbacks which allows to the practitioner to choose the preferred logging method.

Also the search configuration include the verbosity level of the library logger: level 0 is silent, level 1 shows only the objective value at each iteration while level 2 is debug and generates a large amount of logs.

```python
from emlopt.search_loop import SearchLoop
[...]
search = SearchLoop(problem, CONFIG)
search.init_dataset_callback = on_init
search.iteration_callback = on_end_iteration
search.milp_model.solution_callback = on_solution
search.run()
```

Listing C.1: Search callbacks

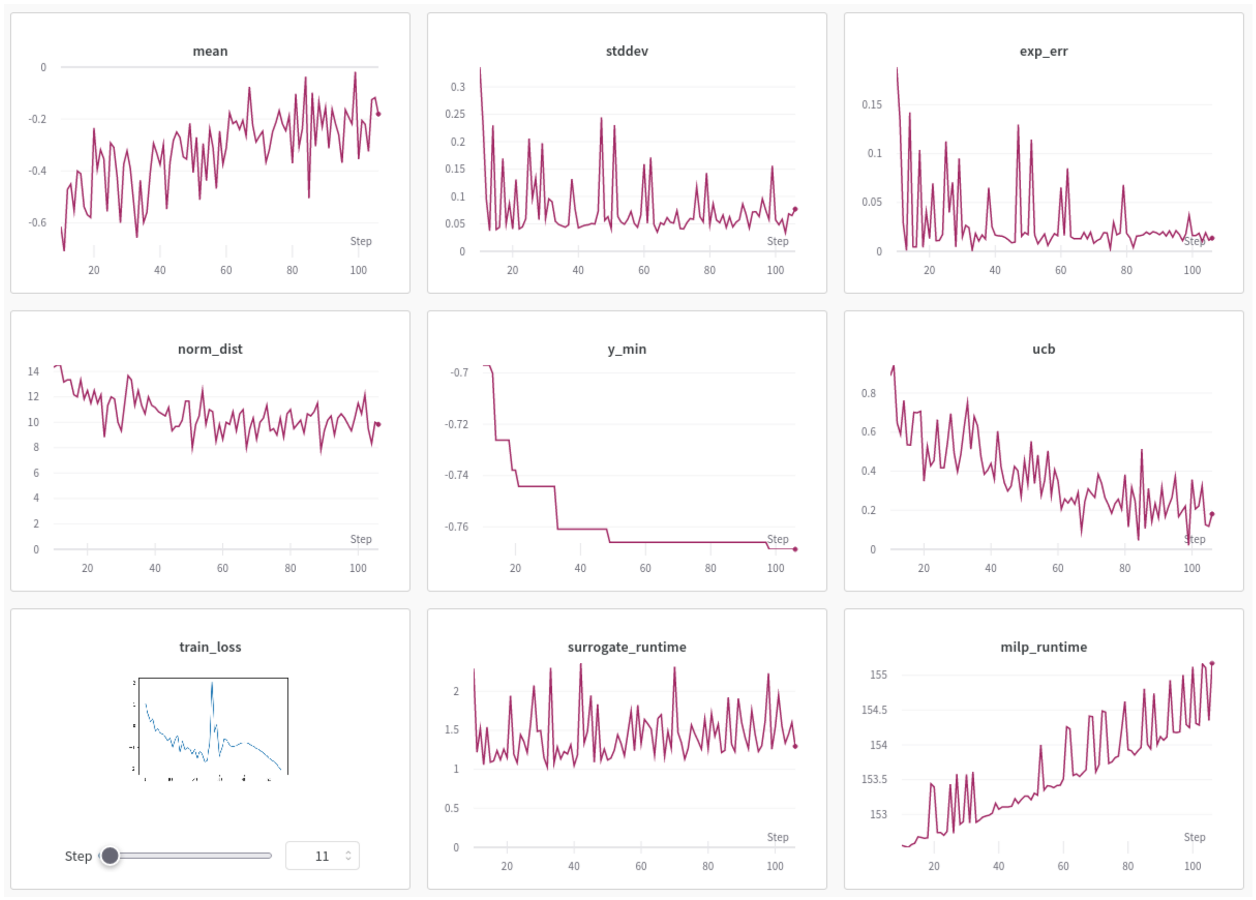


Figure C.1: Metrics collected on WandB dashboard. Having a quick overview is mandatory when debugging high dimensional problems.