SAPIENZA
UNIVERSITÀ DI ROMA

Bayesian Optimization for Hyperparameters Tuning in Neural Networks

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale

Corso di Laurea in Ingegneria Informatica e Automatica

Candidate

Gabriele Onorato
ID number 1979543

Thesis Advisor

Prof. Giampaolo Liuzzi

Academic Year 2023/2024

Bayesian Optimization for Hyperparameters Tuning in Neural Networks
Bachelor's thesis. Sapienza – University of Rome

© 2024 Gabriele Onorato. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: onorato.1979543@studenti.uniroma1.it, gabriele.onorato02@gmail.com

Alla mia nonna materna, Mirella Brilli.

Abstract

This thesis is the culmination of an internal project, equivalent to an internship, that was undertaken during the final year of my degree under the supervision of Prof. Giampaolo Liuzzi.

Bayesian Optimization is a branch of Mathematical Programming that involves the maximization or minimization of a function by selecting input values from a feasible set where membership is easily determined.

This method is commonly employed to black-box functions that require a significant amount of time to evaluate, the number of evaluations that can be performed is quite limited and derivatives are neither used nor available.

For this project, we have chosen Ax as our environment. Ax serves as a wrapper for BOTorch, a framework specifically designed for efficient Bayesian Optimization that utilizes GPU acceleration to enhance performance.

The algorithm devised in this project is engineered to find the global minimum or maximum points of a given function. Initially, it is applied to a selected set of test functions. Subsequently, it is utilized to fine-tune the hyperparameters of a Convolutional Neural Network (CNN). This CNN, trained using PyTorch, is purposed for image classification tasks on the CIFAR10 dataset.

Contents

1 Introduction **1**
- 1.1 Mathematical Optimization . . . 1
- 1.2 Linear Programming . . . 2
  - 1.2.1 Efficiency of the Simplex Algorithm . . . 3
- 1.3 Nonlinear Programming . . . 4
- 1.4 Integer Programming . . . 8

2 Neural Networks **9**
- 2.1 Overview . . . 9
- 2.2 Machine Learning . . . 9
- 2.3 Historical Context . . . 9
- 2.4 Basic Structure of Neural Networks . . . 9
  - 2.4.1 Input Layer . . . 10
  - 2.4.2 Hidden Layers . . . 10
  - 2.4.3 Output Layer . . . 10
- 2.5 Learning Process . . . 10
- 2.6 Types of Neural Networks . . . 11
- 2.7 Applications . . . 12
  - 2.7.1 Computer Vision . . . 12
  - 2.7.2 Natural Language Processing . . . 13
- 2.8 Activation Functions . . . 13
  - 2.8.1 The Problem of Exploding and Vanishing Gradients . . . 14
- 2.9 Loss Functions and Optimizers . . . 15
  - 2.9.1 Loss Functions for Regression . . . 16
  - 2.9.2 Loss Functions for Classification . . . 16
- 2.10 Performance Metrics . . . 17
- 2.11 CIFAR-10 Dataset . . . 19

3 Bayesian Optimization **20**
- 3.1 Overview . . . 20
- 3.2 Gaussian Process Regression . . . 20
  - 3.2.1 GP Posterior on the Objective Function . . . 21
  - 3.2.2 Acquisition Function . . . 21
- 3.3 Mean Function and Kernel . . . 22
  - 3.3.1 Mean Function . . . 22
  - 3.3.2 Kernel (Covariance Function) . . . 22
- 3.4 Acquisition Functions . . . 24
  - 3.4.1 Expected Improvement (EI) . . . 24
  - 3.4.2 q-Expected Improvement (q-EI) in Parallel Evaluations . . . 25
  - 3.4.3 Probability of Improvement (PI) . . . 25

3.4.4 Upper Confidence Bound (UCB) . . . 25
3.5 Model Initialization . . . 26
3.5.1 Sobol' Sequences . . . 26
3.6 Integer Variables in Computer Experiments . . . 27
3.7 Transformations to Smooth the Function . . . 29
3.7.1 Logarithmic Transformation . . . 29
3.7.2 Reciprocal Transformation . . . 29
3.7.3 Comparison and Application . . . 29
3.8 A Basic Pseudo-code for Bayesian Optimization . . . 30
3.8.1 Multi-Objective Optimization and Outcome Constraints . . . 30
3.9 BOTorch and Ax . . . 30
3.9.1 Introduction to BOTorch . . . 30
3.9.2 Introduction to Ax . . . 31
3.9.3 Additional Capabilities of Ax . . . 31
3.9.4 Conclusion . . . 32

4 Experimental Outcomes **33**
4.1 Rosenbrock Banana Function . . . 33
4.2 Neural Network Structure . . . 36
4.2.1 Network Architecture . . . 36
4.2.2 Convolutional Layers . . . 37
4.2.3 Batch Normalization Layer . . . 37
4.2.4 ReLU Activation Function . . . 37
4.2.5 Max Pooling Layer . . . 37
4.2.6 Fully Connected Layer . . . 37
4.2.7 Optimizer and Loss Function . . . 38
4.3 Hyperparameters Overview . . . 38
4.3.1 Number of Layers . . . 38
4.3.2 Neurons per Layer . . . 38
4.3.3 Kernel Size and Stride . . . 38
4.3.4 Activation Functions . . . 39
4.3.5 Optimizer . . . 39
4.4 Chosen Variables . . . 39
4.4.1 Curse of Dimensionality . . . 39
4.4.2 Feature Engineering . . . 39
4.4.3 Utility Function . . . 39
4.4.4 Problem Formalization . . . 40
4.5 First Discoveries . . . 41
4.5.1 Unexpected Event . . . 41
4.5.2 First Results . . . 41
4.6 Improved Experimentation . . . 43
4.6.1 Plot Analysis . . . 47

5 Conclusions **48**

Bibliography **49**

Acknowledgements **51**

Chapter 1

Introduction

1.1 Mathematical Optimization

Mathematical Optimization (also known as Mathematical Programming) is a subfield of Applied Mathematics that consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function.

Optimization problems arise in all quantitative disciplines from computer science and engineering to operations research and economics.

Mathematical Optimization encompasses several types, including but not limited to:

- **Linear Programming**: This involves problems in which the objective function and the constraints are linear.
- **Nonlinear Programming**: This involves problems where the objective function or the constraints or both contain nonlinear parts.
- **Integer Programming**: In this type, some or all of the variables are required to be integers.
- **Stochastic Programming**: This type considers optimization in the presence of uncertainty.
- **Multi-Objective Programming**: This involves optimization problems with multiple conflicting objectives, with solutions that may be evaluated based on Pareto efficiency.

Indeed, only Linear and Nonlinear Programming are mutually exclusive. However, we will encounter problems that simultaneously fall into the categories of Nonlinear, Integer, and Stochastic programming.

1.2 Linear Programming

Linear Programming, henceforth referred to as LP, is a method used for optimizing a linear objective function that is subject to linear equality and inequality constraints. The feasible region of this method is a convex polytope.

LP problems are typically articulated in the following standard form:

$$\begin{cases} \min c^T x \\ Ax = b \\ x \geq 0 \end{cases} \tag{1.1}$$

Numerous practical issues in fields such as Economics and Production can be effectively represented as LP problems, as illustrated in Chapter 3 of Massimo Roma's notes [1], It's also common to find that when modeling a problem, it becomes simpler to represent it in a format known as the "general form" as shown below.

$$\begin{cases} \min c^T x \\ Ax \geq b \end{cases} \tag{1.2}$$

As outlined in Chapter 6.1 of the referenced notes [1], this problem is equivalent to 1.1. Henceforth, we will refer to the latter problem without lack of generality.

Before analysing the Simplex Method, we will illustrate below one of the most important theorem of LP.

Theorem 1.2.1 (Fundamental Theorem of Linear Programming)**.**
Consider the LP problem 1.2, if the polyhedron $P = \{x \in R^n | Ax \geq b\}$ does not contain any line, then only and only one of the subsequent statement is true:

- *The problem 1.2 has no solution/is infeasible, that is, the polyhedron P is empty.*
- *The problem 1.2 is unbounded.*
- *The problem 1.2 has an optimal solution, and at least one of them is a vertex of the polyhedron P.*

The Simplex Method exploits this theorem by sequentially visiting the vertices of P. It applies the sufficient condition for optimality, or determines that the problem is unbounded.

Delving into the details of how the Simplex Method operates is not a minor discussion. It is extensively covered in Chapter 6 of Massimo Roma's notes [1], as well as in numerous other university-level textbooks.

In the following subchapter, we will examine the efficiency of this algorithm from a computer scientist's perspective. This analysis will undoubtedly be beneficial for future comparisons with other algorithms, including the one we designed.

1.2.1 Efficiency of the Simplex Algorithm

The question of whether a pivoting rule exists within the Simplex Method (or Algorithm) that solves the problem in polynomial time remains unresolved. We will commence our analysis by examining the fundamental question: How challenging is a Linear Programming problem?

It has been established that problems characterized by linear inequality constraints and a linear objective function belong to the class P (Polynomial Time). This classification signifies that these problems can be solved in polynomial time with respect to their input size. A notable method for solving such problems is Karmarkar's Algorithm [2], which provides a polynomial-time solution.

Contrarily, the Simplex Algorithm, when employing Bland's rule of pivoting (which guarantees finite termination), has a scenario where it traverses all vertices of the polytope before termination. This process, in fact, exhibits exponential time complexity because the number of vertices V of a polytope P of dimension n with k facets is given by McMullen's Upper Bound Theorem [3]: $O(k^{\lfloor \frac{n}{2} \rfloor})$

Why, then, is the Simplex Algorithm the most used utilized in practice? This is due to the fact that, on average, the Simplex Algorithm functions within a range of $3V$ to $4V$, which is significantly lower, however, the existence of a pivoting rule that ensures polynomial termination in worst-case scenarios remains an unresolved issue in the field of Linear Optimization, and comprehensive examination can be found in V. Klee and G.L. Minty [4].

1.3 Nonlinear Programming

Nonlinear Programming Problems are the most comprehensive type and do not make any assumptions about the linearity of either the objective function or the constraints.

Nonlinear Programming (NLP) Problems are typically modeled as follows:

$$\begin{cases} \min f(x) \\ h(x) = 0 \\ g(x) \leq 0 \end{cases} \tag{1.3}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $h : \mathbb{R}^n \to \mathbb{R}^p$, and $g : \mathbb{R}^n \to \mathbb{R}^m$ are at least once continuously differentiable in the region of interest, and at least twice in case of second order conditions.

Nonlinear Programming is an extensive field. While we don't aim to provide a detailed explanation, we will present the most crucial and recognized findings in this area, the topic is deeply covered in M. Sciandrone's Book [5].

Consider now the unconstrained problem:

$$\begin{cases} \min f(x) \\ x \in \mathbb{R}^n \end{cases} \tag{1.4}$$

Optimality conditions for 1.4 are extensively covered in G. Liuzzi's notes [6] and in M. Sciandrone's book [5].

Numerous tools utilized in Machine Learning belong to this category. For instance, loss functions serve as a measure to assess the accuracy of your algorithm in modeling your dataset. Specifically, the Mean Squared Error, a nonlinear convex function, is employed to adjust weights in Linear Regression. Additionally, constraints are incorporated to prevent overfitting through regularization.

We will only briefly touch upon the most crucial aspects. A key presumption we can make is that **tuning hyperparameters for a Neural Network** undoubtedly constitutes a Nonlinear Programming problem, considering that the function is not analytically accessible and is highly sensitive to alterations.

We are ready to analyse and discuss the Mean Squared Error and the subsequent approaches to achieve results of the following problem:

$$\begin{cases} \min L(w) = \frac{1}{n}||Xw - y||^2 \\ X \in \mathbb{R}^{n \times p} \\ w \in \mathbb{R}^p \\ y \in \mathbb{R}^n \end{cases} \tag{1.5}$$

In this context, n denotes the total number of samples within the dataset, while w signifies a vector consisting of p weights. The term y relates to our target which our model must predict, and our objective is to minimize the error associated with it.

An analytical method for addressing this challenge involves calculating the gradient with respect to w, expressed as:

$$\nabla_w L(w) = \frac{2}{n} X^T (Xw - y)$$

Following this, we set the gradient equal to zero, disregarding any constants, resulting in:

$$X^T(Xw^* - y) = X^T X w^* - X^T y = 0$$

Upon rearranging the elements, we obtain:

$$X^T X w^* = X^T y$$

By multiplying from the left by the inverse of $X^T X$, we deduce:

$$w^* = (X^T X)^{-1} X^T y$$

Here, $(X^T X)^{-1} X^T$ represents the Moore-Penrose pseudoinverse, which identifies a specific solution for the system of linear equations derived from setting the gradient to zero.

Notably, $L(w)$ is classified as a convex function, as its second-order derivative, $\nabla_w^2 L(w) = X^T X$, conforms to the Sylvester criterion, rendering it a positive semidefinite matrix. Consequently, each minimum point is inherently a global minimum.

Although the methodology appears straightforward, computing the pseudoinverse is practically demanding and not universally applicable to general NLP problems. Within this discourse, we explore the most acclaimed strategy for tackling Nonlinear Programming, which involves utilizing first-order derivatives.

Algorithm 1: Gradient Descent

```
Data: a starting point x_0 ∈ ℝ^n, a function f(x) to minimize
η ← a constant small value
for k = 0, 1, 2, ... do
    if ∇f(x_k) = 0 then
        return x_k
    x_{k+1} ← x_k − η∇f(x_k)
end
```

In the scenario presented, the parameter η is assigned a small value based on the presumption that $\nabla f(x)$ maintains at least Lipschitz continuity, a concept examined in Chapter 16 of G. Liuzzi's Notes [6]. Under these hypotheses, the algorithm has proven convergence towards points of minima.

Another widely recognized approach, built on the principles of gradient descent, is extensively employed in the field of machine learning. This method is predicated on the notion that computing the full gradient across a large dataset is expensive. Therefore, it segments the dataset into smaller batches. The algorithm iteratively processes these batches and designates an epoch upon completing a full pass through the dataset.

Algorithm 2: Stochastic Gradient Descent

```
Data: a starting value of weights w_0 ∈ R^p, a function L(w) to minimize
η ← a constant small value
for k = 0, 1, 2, ..., N do
    for i = 0, 1, 2, ...M do
        G_{k,i} ← ∇_i L(w_{k,i}) /* gradient evaluated on the i-th batch */
        if G_{k,i} = 0 then
            return w_{k,i}
        w_{k,i+1} ← w_{k,i} − η G_{k,i}
    end
    w_{k+1} ← w_k
end
return w_N
```

This evaluation method incurs an error proportional to the square root of the batch size. Assuming that the cost of evaluating a batch increases linearly, adopting this approach is particularly beneficial as it is still compatible with regularization techniques. However, it does not guarantee certain convergence, leading to the development of various modified versions. One notable modification is the addition of finite termination conditions, exemplified by D.P. Kingma and J.L. Ba's Adam Optimizer [7]. This optimizer, which we will explore further in the context of our neural network implementation, remains one of the most popular methods in machine learning today, along with its variants (AdaMax, AdaGrad).

Numerous other methods also warrant discussion, particularly those deriving from the Newton-Raphson method, which leverages second-order derivatives to hasten convergence towards points where the gradient vanishes. These methodologies receive thorough treatment in G. Liuzzi's Notes [6] and M. Sciandrone's Book [5]. Notably, Quasi-Newton methods, which rely solely on first-order derivatives, are recognized as some of the fastest algorithms in their class. We plan to integrate these methods into our Bayesian Optimization Loop subsequently.

Additionally, it's important to highlight methods that operate without employing derivatives. These are especially beneficial in scenarios characterized by noisy evaluations or when the gradient cannot be consistently calculated at every iteration. Some of these methods utilize the gradient merely as a termination criterion, while others bypass its use entirely. Specifically, **Bayesian Optimization stands out as a derivative-free strategy that imposes no prerequisites on the objective function's differentiability.** M. Sciandrone's book provides a comprehensive exploration of these derivative-independent methods [5].

We conclude our analysis by considering the complexity inherent in Nonlinear Programming (NLP). It is widely acknowledged that when the termination condition $\nabla f(x) = 0$ is applied, the problem falls within the NP class. Additionally, certain techniques lack proven convergence and are predominantly heuristic in nature, such as the Nelder-Meade method, with others potentially failing to terminate correctly.

Recent research has adopted an innovative approach for conducting worst-case analyses under broadly defined assumptions. A particularly comprehensive study by E.G. Birgin et al. [8], focused on unconstrained NLP problems 1.4, exemplifies this trend. The research illustrates that, with the termination criterion $||\nabla f(x)|| \leq \epsilon$, first-order methods exhibit a worst-case complexity of $O(\epsilon^{-2})$. Furthermore, the analysis has been extended to encompass p-order methods, assuming they adhere to specific regularity conditions.

1.4 Integer Programming

Our discussion on Integer Programming will be concise, given its comprehensive treatment in M. Roma's Notes [1]. Integer programming encompasses a range of problems where variables are restricted to discrete values, rather than continuous ones. This can apply to scenarios such as determining the number of workers in a factory, the daily production quantities of sellable goods, or the total number of factories. Specifically, in our context, it pertains to setting **the number of neurons per layer in a Neural Network.**

We will now delve into the concept known as the relaxation of an Integer Programming problem, formalized as follows:

$$\begin{cases} \min f(x) \\ x \in D \subset \mathbb{R}^n \\ x \in \mathbb{Z}^n \end{cases} \tag{1.6}$$

The *Continuous Relaxation* of this problem removes the integer constraint, meaning the condition $x \in \mathbb{Z}^n$ is omitted. This adaptation opens the door to various insights and analytical perspectives, which are thoroughly discussed in Chapter 9.2 of M. Roma's Notes [1].

One of the quintessential examples of Integer Programming, specifically within the linear scope, is the Binary Knapsack Problem. It has been established that there is a polynomial-time reduction from the SAT (Boolean satisfiability problem) to the Binary Knapsack Problem, thereby classifying it within the NP-Complete category. Consequently, as this specific Integer Programming challenge is NP-Complete, the entire class is deemed NP-Hard.

The range of methods for solving Integer Programming (IP) problems varies significantly. At one end of the spectrum is Total Enumeration, which entails assessing every possible feasible solution – an approach that falls squarely within the EXPTIME (exponential time) complexity class. At the other end are Branch and Bound techniques, which segment the original problem into manageable subsets. For each subset, a permissible solution is sought, enabling the elimination of subproblems that cannot yield superior outcomes. This strategy significantly reduces the required evaluations. However, it's important to note that in the worst-case scenario, the computational time can still be exponential.

The final widespread method involves obtaining an approximate solution via solving the Continuous Relaxation of the original problem and subsequently rounding it to the closest admissible integer solution. It has been established that, particularly in scenarios with specific constraints (even if linear), this strategy can result in solutions that are infeasible or significantly worse than the optimal, deviating considerably from the intended approximation. Therefore, this method is only viable under certain regularity conditions of the problem's bounds. **We plan to adopt this strategy within our Bayesian Optimization algorithm, provided these conditions are met.**

These methods are extensively treated in Chapter 10 of M. Roma's Notes [1].

Chapter 2

Neural Networks

2.1 Overview

In this chapter, we delve into the essence of Neural Networks as applied in Machine Learning, aiming to establish a solid foundation by exploring their fundamental principles, architecture, and capabilities. Subsequently, we will explore their broad applications and discuss their implementation in PyTorch [12].

2.2 Machine Learning

Machine learning is a subset of artificial intelligence that involves the use of algorithms and statistical models to enable computers to perform specific tasks without explicit instructions. It relies on patterns and inference derived from data to make decisions and predictions. Essentially, machine learning processes vast datasets to 'learn' and evolve based on experience. Its applications are wide-ranging and transformative, encompassing areas like speech and image recognition, as well as predictive analytics.

2.3 Historical Context

The idea of neural networks dates back to the 1940s when Warren McCulloch and Walter Pitts presented a mathematical model of a neural network. In the 1950s and 1960s, pioneers like Frank Rosenblatt and Bernard Widrow further developed the concept with models like the Perceptron and Adaline, which laid the foundation for modern neural networks. Despite early enthusiasm, limitations in technology and theoretical understanding led to periods of reduced interest, known as "AI winters". The resurgence of neural networks in the late 20th and early 21st centuries has been fueled by advances in computational power, the availability of large datasets, and improvements in algorithms, leading to the current era of deep learning.

2.4 Basic Structure of Neural Networks

A neural network consists of layers of interconnected nodes or neurons, where each node represents a specific output function called an activation function. The simplest structure includes three layers: an input layer, at least one hidden layer, and an output layer. Neurons in each layer connect to every neuron in the subsequent layer, forming a dense network through which data flows and is processed.

2.4.1 Input Layer

The input layer serves as the gateway for raw data, with each neuron representing a distinct input feature. In our scenario, where the input is an image, the number of neurons will be determined by the formula $height \times width \times channels$, reflecting the image's dimensions and color depth. The exact number of channels is contingent on the input format.

2.4.2 Hidden Layers

Hidden layers execute computations on inputs from the preceding layer, employing weights (parameters) and biases that are refined throughout the training process by minimizing a loss function. The complexity and depth of patterns a network can learn are significantly influenced by the number of hidden layers and the neurons within each layer. **Our Bayesian Optimization algorithm will utilize the number of neurons per layer as an input parameter**.

2.4.3 Output Layer

The output layer produces the final prediction or classification result, depending on the task the network is designed to perform.

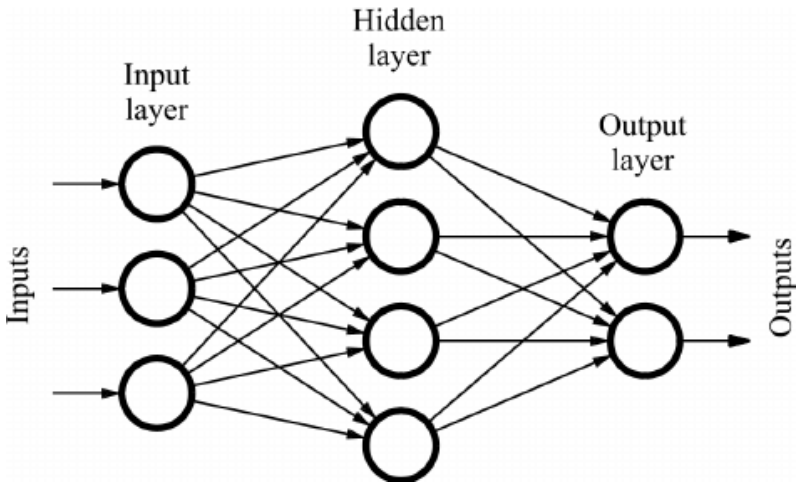


Figure 2.1. Example of a simple Feedfoward Neural Network

2.5 Learning Process

Neural networks learn through a process known as backpropagation. During training, the network makes predictions on the input data, calculates the error of its predictions compared to the actual target values, and adjusts its weights and biases to minimize this error. The optimization of weights is typically performed using gradient descent or variations thereof.

2.6 Types of Neural Networks

There are various types of neural networks, each suited to different tasks and data types. Some of the most common include:

- **Feedforward Neural Networks (FNN):** The simplest type, where connections between the units do not form a cycle as illustrated in (Figure 2.1).
- **Convolutional Neural Networks (CNN):** Exceptionally suited for image-related tasks, CNNs employ convolutional layers to extract spatial features, use max pooling for dimensionality reduction, and subsequently channel the processed data through a series of fully connected layers for classification purposes.

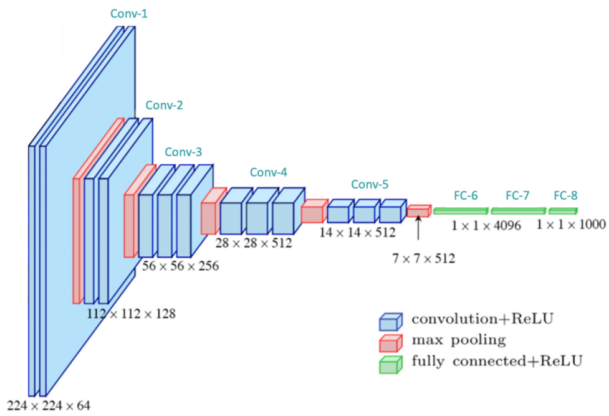


Figure 2.2. A CNN with many convolutional and fully connected layers.

- **Recurrent Neural Networks (RNN):** Designed for sequential data, such as time series or natural language, with connections forming directed cycles that uses the recently generated data to predict the next output.

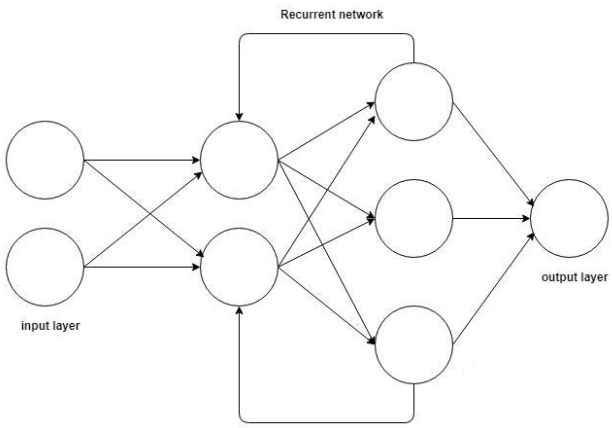


Figure 2.3. A RNN example with two cycles.

- **Autoencoders:** Used for unsupervised learning tasks, such as dimensionality reduction or feature learning, by learning to compress and then reconstruct the input data.
- **Generative Adversarial Networks (GANs):** Composed of two networks, a generator and a discriminator, competing against each other to generate new data samples.

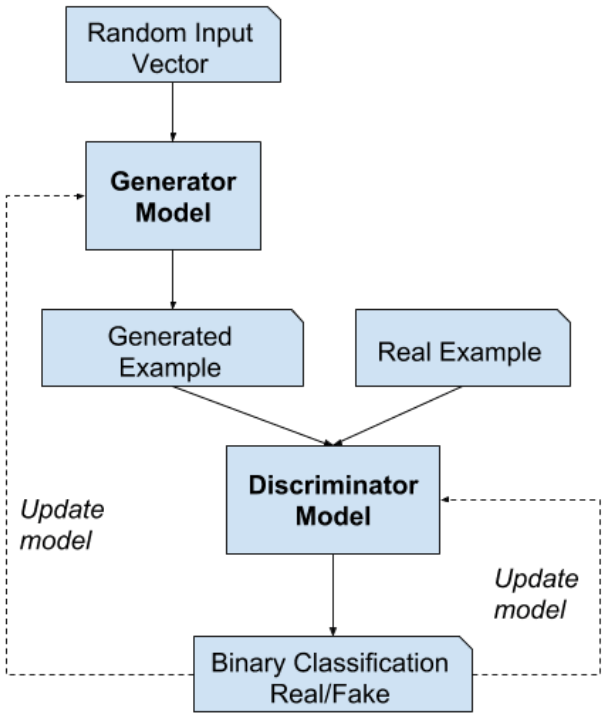


Figure 2.4. A GAN that evaluates whether an input example is generated or real, and subsequently updates both models accordingly.

2.7 Applications

Neural networks have revolutionized a myriad of domains, spanning computer vision, speech recognition, natural language processing, healthcare, financial forecasting, among others. Their capacity to extrapolate insights from data, enabling predictions or classifications, has established them as a pivotal element in the advancement of contemporary artificial intelligence and machine learning frameworks.

2.7.1 Computer Vision

CNNs have been fundamental to the advancement of computer vision technologies, enabling groundbreaking applications such as facial recognition, object detection, and image classification. Our project focuses on **tuning hyperparameters** for a CNN designed explicitly for image classification, with the objective of elevating its precision and efficiency through the application of various metrics.

2.7.2 Natural Language Processing

RNNs and the more recent Transformer architectures have significantly advanced machine understanding and generation of human language. These technological strides have opened new horizons in machine translation, text summarization, and sentiment analysis. Among the most prominent Large Language Models (LLMs), ChatGPT utilizes these sophisticated technologies, enhancing productivity across various industrial sectors and daily activities.

2.8 Activation Functions

An activation function in neural networks serves as a crucial mathematical tool that injects non-linearity into the network's learning mechanism. It essentially takes the input signal and transforms it into an output signal for the subsequent layer. This step is vital for the network's ability to digest and learn from complex data patterns, something that mere linear operations cannot accomplish.

Without the introduction of non-linearity, a neural network with many layers would be equivalent to a single-layer perceptron, limited to understanding only linear relationships. Simply put, layering multiple linear transformations together would still yield a linear outcome. Non-linear activation functions shatter this barrier, equipping neural networks with the capability to capture and model the complex, nuanced relationships found within various types of data, such as images, audio, and text.

The selection of activation functions is diverse, and choosing the appropriate type for each layer is a critical decision. This decision significantly influences the network's performance and is **an example of a hyperparameter that can be tuned using our Bayesian Optimization Algorithm**.

Let's plot and analyze some well-known activation functions, examining their features and gradients.

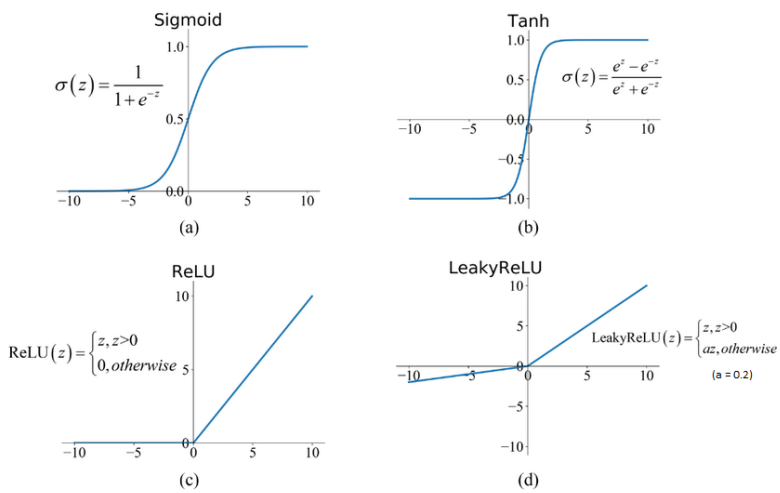


Figure 2.5. The four most commonly used activation functions are showed here, with LeakyReLU featuring a as a hyperparameter.

One of their most notable characteristics is that the gradients of all these functions are defined by combinations of the functions themselves, as illustrated below:

- Sigmoid: $\nabla\sigma(z) = \sigma(z)(1 - \sigma(z))$
- Hyperbolic Tangent: $\nabla \tanh(z) = 1 - \tanh^2(z)$
- ReLU and LeakyReLU:

$$\nabla \text{ ReLU}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 \text{ or } a & \text{otherwise} \end{cases}$$

his characteristic simplifies the evaluation of gradients, even in complex networks, as it provides an analytical solution without approximations. A comprehensive survey and benchmark of these features is detailed in S.R Dubey, et al. [9].

2.8.1 The Problem of Exploding and Vanishing Gradients

As mentioned in the previous chapters, the training phase involves the systematic updating of weights via backpropagation. This process is fundamentally about computing the gradient of the loss function with respect to each weight, employing the chain rule. These computed gradients are subsequently utilized by the selected optimizer to refine the model's parameters.

A critical challenge often encountered is the vanishing gradient problem, which is particularly prevalent in deep neural networks comprising numerous layers. The essence of this problem lies in the gradients of the network's weights diminishing towards zero. As a result, the network struggles to effectively adjust its weights, which drastically slows down the training process or may even stall it altogether. This problem is typically exacerbated by the use of activation functions such as the Sigmoid or Hyperbolic Tangent, which tend to saturate.

To mitigate this issue, we have opted to incorporate the Rectified Linear Unit (ReLU) as our activation function within the neural network. The ReLU function is known for its ability to maintain a non-zero gradient, thereby facilitating a more robust and continuous learning process during our **Bayesian Optimization Loop**. This strategic choice aims to enhance the efficiency of our training process and improve the overall performance of the neural network.

To prevent exploding gradients, methods such as the use of gradient rescaling and L1 and L2 penalty functions are employed, drawing on concepts similar to those found in Lagrange multipliers, as shown below.

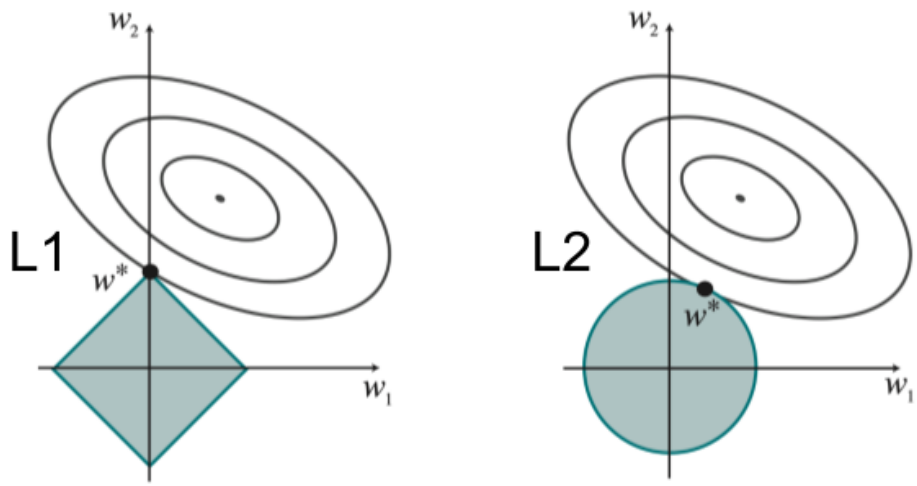


Figure 2.6. L1 and L2 regularization techniques

Regularization alters the optimization domain by imposing constraints that shift the position of the minimum point within the solution space. For instance, L1 regularization tends to drive smaller weights toward zero, influencing the model's complexity and robustness.

These methods are also effective in preventing overfitting. For a detailed analysis, refer to the research by Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio [10].

2.9 Loss Functions and Optimizers

In training a neural network, the loss function plays a pivotal role by evaluating the discrepancy between the network's current predictions and the actual ground truth within the dataset at each step. This evaluation guides the estimation of necessary adjustments. To optimize this process, first-order derivative methods, such as SGD (Alg 2) are commonly employed. These methods leverage backpropagation to calculate the gradient of the loss function accurately, followed by to adjusting the weights accordingly. For our implementation, we have opted for the Adam Optimizer [7], as implemented in PyTorch [12], due to its proven efficiency and effectiveness in handling such optimization tasks.

Indeed, the choice of an optimizer is crucial for developing an efficient neural network. And **it stands as a selectable hyperparameter within our Bayesian Optimization Algorithm**.

The selection of the Loss Function is typically made on a case-by-case basis, influenced primarily by engineering considerations tailored to the specific application at hand. We will outline the most commonly used methods, but for a more detailed discussion on loss functions, refer to the initial subchapters of each section in the work of Juan Terven et al. [11]

2.9.1 Loss Functions for Regression

We will analyze the two most common loss functions for regression: Mean Squared Error (MSE) and Mean Absolute Error (MAE), assuming that n is the number of predictions, x are the features in input, $h(x)$ represents the vector of predictions, and y denotes the ground truth.

Regarding the MSE which has been briefly introduced in 1.5, it is defined as follows:

$$MSE(x) = \frac{1}{n}||h(x) - y||^2 \tag{2.1}$$

Here, the MSE is characterized by its differentiability, allowing for easy computation of gradients due to its lack of discontinuities. In contrast, the MAE has a gradient that is undefined at zero, making it less straightforward for gradient-based optimization methods.

$$MAE(x) = \frac{1}{n}\sum_{i=1}^{n}|h(x_i) - y_i| \tag{2.2}$$

Both of them are widely used, with MSE particularly emphasizing larger errors due to its squared term.

2.9.2 Loss Functions for Classification

In classification tasks, it's more challenging to establish a straightforward scale for evaluating error compared to regression tasks. For instance, in a simple binary classification task, the outcome is either correct or incorrect. So, how can we define the error in such cases? Let's explore the most commonly used methods starting from binary classification:

Binary Cross-Entropy Loss (BCE), also known as Log Loss, quantifies the divergence between the predicted probability, p, of a class being 1 and the actual label, y, where y can be either 0 or 1. This measure is used to evaluate the accuracy of a binary classification model. Typically, a threshold of 0.5 is employed to decide whether the predicted probability is categorized as class 1 or class 0.

$$BCE(y, p) = -(y\log(p) + (1 - y)\log(1 - p)) \tag{2.3}$$

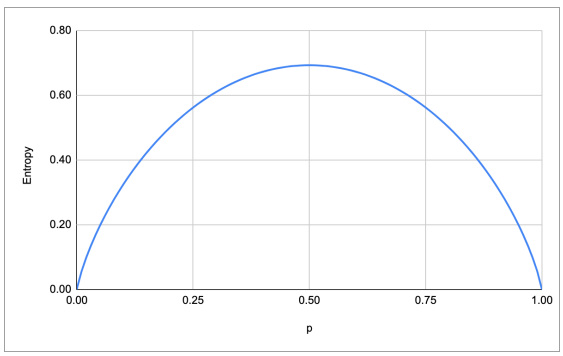


Figure 2.7. Binary Cross Entropy Loss

BCE can be readily adapted to incorporate weights, particularly useful in scenarios where the sample distribution is imbalanced. This adaptation involves assigning a specific weight to each sample as follows:

$$BCE(y, p, w) = -w_i(y \log(p) + w_i(1 - y) \log(1 - p)) \tag{2.4}$$

Weighted Binary Cross-Entropy Loss (WBCE) adjusts the model's focus by assigning greater importance to the under-represented class in imbalanced datasets. This weighting approach enhances the model's sensitivity to minority classes, improving performance on challenging datasets.

Binary Cross-Entropy Loss (BCE) can be effectively extended to multi-class classification tasks by introducing Categorical Cross-Entropy Loss (CCE):

$$CCE(p, y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} y_{i,j} \log(p_{i,j}) \tag{2.5}$$

Here, N represents the number of samples, and C denotes the number of classes involved.

This adaptation enables the model to handle multiple classes by calculating the loss for each sample based on the discrepancy between the predicted probabilities and the actual class labels, and then averaging this loss across the entire dataset.

Typically, the true label y is represented as a one-hot encoded vector, although in some implementations, the classes may be denoted by integers.

For our objectives, we utilized the Categorical Cross-Entropy Loss, as implemented in PyTorch. [12]

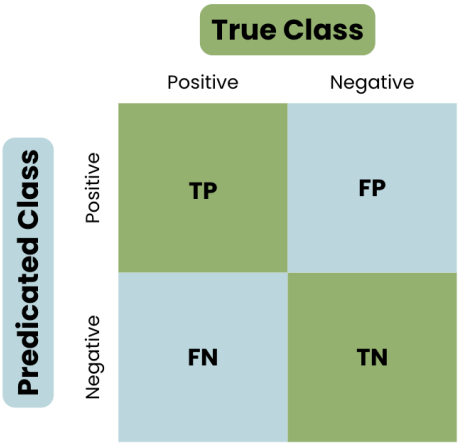
2.10 Performance Metrics

Performance metrics are essential for assessing and fine-tuning neural networks, providing insights into various aspects of model accuracy and effectiveness.

The most used metrics for classification purposes are:

- **Accuracy**: measures the proportion of correct predictions made by the model across all classes, offering a high-level view of performance.
- **Specificity and Sensitivity**: Specificity measures the capacity to correctly detect true negatives, whereas sensitivity assesses the model's ability to correctly identify true positives.
- **Precision and Recall**: are critical for tasks where the distinction between different types of errors is significant; precision measures the accuracy of positive predictions, while recall assesses how well the model identifies actual positives.
- **F1 Score**: combines these two metrics, giving a balanced measure of a model's precision and recall, especially useful in imbalanced datasets. The formula is $\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

- **Confusion Matrix**: is a handy visualization tool that shows the counts of actual versus predicted classifications, providing clear insight into the model's performance across different categories.



$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Figure 2.8. Confusion Matrix and Precision Recall formulas

- **Receiver Operating Characteristic** (ROC) curve and its corresponding **Area Under the Curve** (AUC): measure a model's ability to discriminate between classes at various threshold levels, with a higher AUC indicating better model performance.

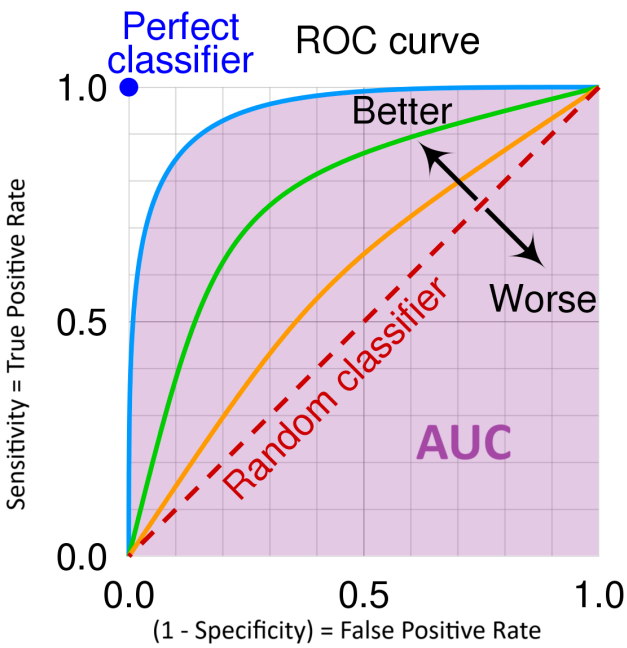


Figure 2.9. ROC curve and AUC (in purple)

For regression tasks or models predicting continuous outputs, Mean Squared Error (2.1) and Mean Absolute Error (2.2) are used to quantify the average magnitude of the errors in predictions.

We will focus on **maximizing Accuracy as our objective function in the Bayesian Optimization Loop**. For an extensive discussion of these metrics, see the work of Juan Terven, et al. [11].

2.11 CIFAR-10 Dataset

The CIFAR-10 dataset is a widely used benchmark in the field of machine learning for evaluating image recognition algorithms. It consists of 60,000 RGB color images, each of size 32x32 pixels, categorized into 10 classes with 6,000 images per class. The classes represent everyday objects such as airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is divided into a training set of 50,000 images and a test set of 10,000 images. Developed by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton [13]. CIFAR-10 is popular for its manageable size and complexity, making it an excellent resource for testing algorithms quickly and effectively in computer vision research.

Our neural network is trained on this dataset with the goal of maximizing the average accuracy on the test set. Although our project focuses primarily on this metric, Ax [20] also supports multi-objective optimization and outcome constraints, allowing for consideration of metrics such as F1 score and AUC as well as class-specific accuracy, even though they are not utilized in our implementation. **We will delve into these details later when discussing our Bayesian Optimization Algorithm**.

Figure 2.10. Example Classes of the CIFAR-10 Dataset

Chapter 3

Bayesian Optimization

3.1 Overview

We now explore the core optimization technique utilized in our project: Bayesian Optimization (BOpt). BOpt is a probabilistic, model-based approach designed to find the optimal solution of expensive-to-evaluate functions. It effectively handles stochastic noise in function evaluations by creating a surrogate model for the objective function and quantifying the uncertainty in that surrogate using Gaussian process regression, a Bayesian machine learning technique. This surrogate model is then used to derive an acquisition function that determines the next sampling points. BOpt is particularly suited for solving problems such as the following:

$$\begin{cases} \max f(x) \\ x \in A \end{cases} \tag{3.1}$$

Typically, the following properties are desired:

- The input $x \in \mathbb{R}^d$ should not be excessively large; typically, a dimensionality of $d \leq 20$ is most suitable for practical applications. We will later discuss how the input dimensionality impacts the efficiency of the optimization process.
- The feasible set A is a simple set in which it is easy to assess membership, such as a hyper-rectangle or a d-dimensional simplex. This assumption can be relaxed, but it is not within the scope of our work to cover more complex sets.
- More importantly, $f(x)$ lacks known special structures such as convexity or linearity, is expensive to evaluate, and is at least continuous.
- When we choose the next point at which $f(x)$ will be evaluated, we only observe its value. Neither first nor second-order derivatives are used, giving it a 'derivative-free' property. Additionally, the function can be obscured by stochastic noise.

We will now present the most prominent and useful insights from the analysis in Peter I. Frazier's tutorial paper [14].

3.2 Gaussian Process Regression

Gaussian Process (GP) Regression is a Bayesian statistical approach for modeling functions. It is based on two key concepts: the GP posterior on the objective function and the Acquisition Function.

3.2.1 GP Posterior on the Objective Function

The GP posterior provides a probabilistic framework for estimating the objective function. It combines prior beliefs with observed data to create a Gaussian posterior distribution, offering both a mean estimate and an uncertainty measure (variance). This approach allows us to incorporate prior knowledge and update our understanding as more data points are observed.

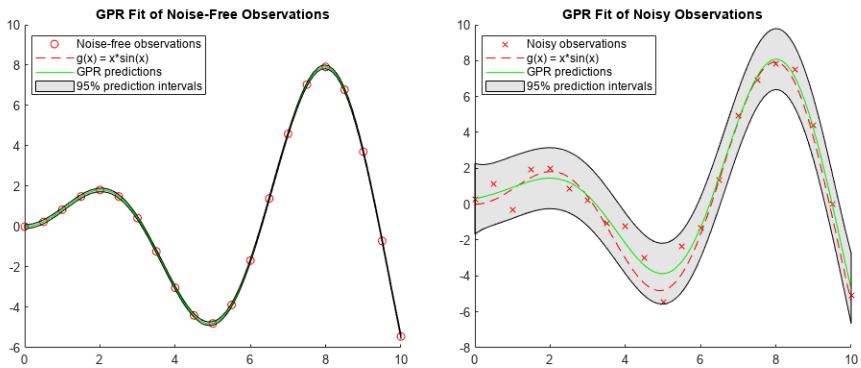


Figure 3.1. GP Regression in both noisy and noise-free observations

3.2.2 Acquisition Function

The acquisition function is a crucial component in GP regression. It guides the selection of the next points for evaluation by balancing exploration and exploitation. The function uses the posterior distribution to determine where to sample next, in order to find the global optimum of the objective function efficiently.

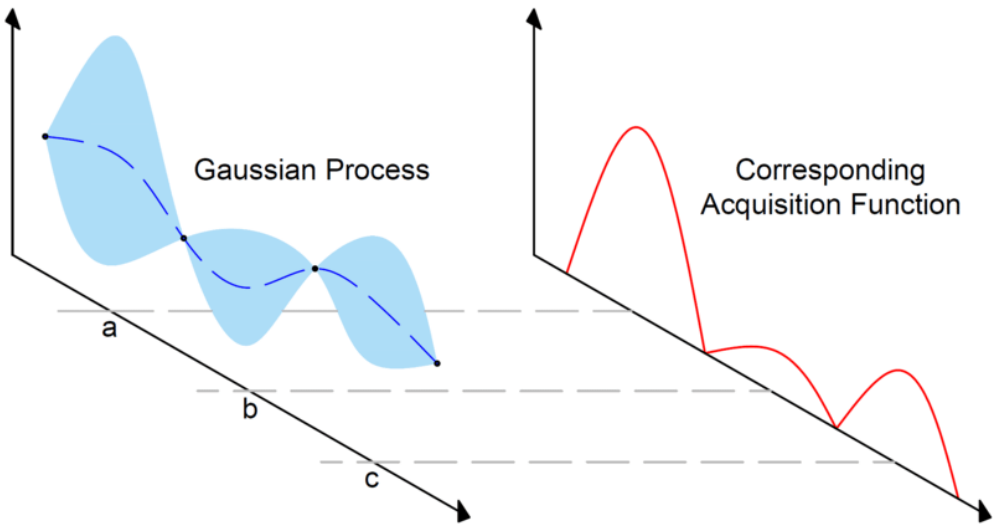


Figure 3.2. GP Regression and its Acqusition Function

A visual and interactive exploration of how these concepts work can be found in the work of Görtler et al. [15].

3.3 Mean Function and Kernel

In Gaussian Process (GP) Regression, the choice of the mean function and kernel (also known as the covariance function) is crucial as they define the properties of the functions we aim to model.

3.3.1 Mean Function

The mean function represents the expected value of the function we are modeling. There are several types of mean functions that can be chosen based on the prior knowledge about the function.

Constant Mean A constant mean function, $\mu_0(x) = \mu$, assumes that the function has a constant mean across its domain. This is the simplest form of mean function and is often employed when there is no prior information suggesting a different mean structure, making it a common choice.

Parametric Mean Function A parametric mean function uses a parametric form to model the mean of the function, allowing the incorporation of prior knowledge about the general trend of the function. It is typically expressed as:

$$\mu_0(x) = \mu + \sum_{i=1}^{p} \beta_i \Psi_i(x)$$

where Ψ_i are parametric functions, usually low-order polynomials. This approach provides flexibility in representing more complex mean structures.

3.3.2 Kernel (Covariance Function)

The kernel defines the covariance between function values at different points and encodes assumptions about the function's smoothness, periodicity, and other properties. Two commonly used kernels in GP regression are the Gaussian kernel and the Matérn kernel.

Gaussian (RBF) Kernel The Gaussian kernel, also known as the Radial Basis Function (RBF) kernel, assumes that the function is infinitely differentiable, leading to a very smooth function. It is defined as:

$$\Sigma_0(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||^2}{2l^2}\right)$$

where l is the length-scale parameter that controls the smoothness of the function. The Gaussian kernel is widely used due to its simplicity and the strong smoothness assumption it imposes.

Matérn Kernel The Matérn kernel is more flexible than the Gaussian kernel and can model functions with varying degrees of smoothness. It is defined as:

$$\Sigma_0(x_i, x_j) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}||x_i - x_j||}{l} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}||x_i - x_j||}{l} \right)$$

where ν controls the smoothness of the function and K_ν is the modified Bessel function. As ν increases, the Matérn kernel becomes more similar to the Gaussian kernel.

Choosing the right mean function and kernel is crucial for accurately modeling the objective function in GP regression. Constant and parametric mean functions offer flexibility in representing mean behavior, while Gaussian and Matérn kernels impose different assumptions on the function's smoothness and properties.

In our work, the choice of the mean function and kernel will be made using Ax [20], our Bayesian Optimization environment, which utilizes implementations from BOTorch [17].

3.4 Acquisition Functions

Acquisition functions are essential in Bayesian Optimization, directing the selection of the next points for evaluating the objective function. Their evaluations are computationally inexpensive, allowing the use of optimization methods such as Gradient Descent and BFGS, which utilize first-order derivatives. In the following, we discuss some commonly used acquisition functions.

3.4.1 Expected Improvement (EI)

Expected Improvement (EI) is one of the most popular acquisition functions. It builds upon the concept of improvement, which is defined as the amount by which a new observation $f(x)$ exceeds the current best observation $f(x^+)$, if it does:

$$I(x) = \max(f(x) - f(x^+), 0)$$

EI calculates the expected value of this improvement, taking into account both the mean and the uncertainty of the predictions. The EI at a point x is defined as:

$$\mathrm{EI}(x) = \mathbb{E}[I(x)] = \mathbb{E}[\max(f(x) - f(x^+), 0)]$$

The closed form of EI, under the common assumption that the prediction follows a Gaussian distribution, is as follows:

$$\mathrm{EI}(x) = (\mu(x) - f(x^+))\Phi\left(\frac{\mu(x) - f(x^+)}{\sigma(x)}\right) + \sigma(x)\phi\left(\frac{\mu(x) - f(x^+)}{\sigma(x)}\right)$$

where $\mu(x)$ is the predicted mean, $\sigma(x)$ is the predicted standard deviation, $\Phi(\cdot)$ is the cumulative distribution function, and $\phi(\cdot)$ is the probability density function of the standard normal distribution.

By considering the expected improvement, the EI tends to favor points with high uncertainty if the potential improvement is significant, effectively balancing exploration and exploitation.

The gradient of EI, can be derived as follows. Let $\mu(x)$ and $\sigma(x)$ be the predicted mean and standard deviation at point x, respectively. The EI gradient is given by:

$$\nabla\mathrm{EI}(x) = \nabla\sigma(x)\left(\frac{(f(x^+) - \mu(x))\phi(z) + \sigma(x)\Phi(z)}{\sigma(x)}\right)$$

where $\phi(z)$ and $\Phi(z)$ are the probability density function and the cumulative distribution function of the standard normal distribution, respectively, and $z = \frac{f(x^+)-\mu(x)}{\sigma(x)}$. The terms $\nabla\mu(x)$ and $\nabla\sigma(x)$ are the gradients of the predicted mean and standard deviation, respectively.

The gradient of EI is available in closed form when the Gaussian Process (GP) model used has a closed-form expression for the mean and covariance functions. This typically occurs when the kernel function of the GP is chosen so that its derivatives are analytically tractable. Common kernels like the Gaussian (RBF) kernel and the Matérn kernel with specific parameters often allow for such closed-form solutions, enabling efficient computation of the EI gradient.

3.4.2 q-Expected Improvement (q-EI) in Parallel Evaluations

In many practical scenarios, it is desirable to evaluate multiple points in parallel to speed up the optimization process. Expected Improvement can be extended to support parallel evaluations through various strategies.

One common approach is the use of the "q-Expected Improvement" (q-EI), which generalizes EI to the setting where q points are evaluated simultaneously. The q-EI is defined as the expected improvement over the best observed value, considering the joint distribution of improvements at q points. This joint consideration helps in capturing the dependencies between the evaluations, providing a more robust parallel evaluation strategy. The q-EI at a set of points $\mathbf{x}_{1:q}$ is defined as:

$$\text{q-EI}(\mathbf{x}_{1:q}) = \mathbb{E}\left[\max\left(\max_{i=1,\ldots,q} f(x_i) - f(x^+), 0\right)\right]$$

where $\mathbf{x}_{1:q} = \{x_1, x_2, \ldots, x_q\}$ are the points to be evaluated in parallel, and $f(x^+)$ is the best observed value so far.

With these modifications, EI can effectively support parallel evaluations, thereby enhancing the efficiency of the Bayesian Optimization process in scenarios where multiple evaluations are conducted simultaneously.

3.4.3 Probability of Improvement (PI)

Probability of Improvement (PI) focuses on the likelihood of improving over the current best observation. It is defined as:

$$\text{PI}(x) = \mathbb{P}(f(x) > f(x^+))$$

PI tends to exploit more by selecting points with a high probability of improving the current best value, but it may overlook regions of high uncertainty, making it more suited for local search.

3.4.4 Upper Confidence Bound (UCB)

The Upper Confidence Bound (UCB) acquisition function balances exploration and exploitation by considering both the mean and the variance of the predictions. It is defined as:

$$\text{UCB}(x) = \mu(x) + \kappa\sigma(x)$$

where κ is a parameter that controls the balance between exploration and exploitation. Higher values of κ encourage exploration by giving more weight to the uncertainty term.

The gradient of the UCB function can be derived as follows:

$$\nabla\text{UCB}(x) = \nabla\mu(x) + \kappa\nabla\sigma(x)$$

where $\nabla\mu(x)$ and $\nabla\sigma(x)$ are the gradients of the predicted mean and standard deviation, respectively. This gradient is useful for optimization algorithms that rely on gradient information to efficiently explore the search space.

Our algorithm will primarily utilize UCB, and alternatively EI, as their gradients are available in closed form for sequential evaluations. Since our environment involves the task of training a neural network, which is inherently sequential, this allows us to significantly speed up the process. Additionally, we will discuss other scenarios where parallel evaluations are highly advantageous.

An extensive treatment of acquisition functions and Gaussian Processes (GPs) can be found in the works of Donald R. Jones, Matthias Schonlau, and William J. Welch [16], as well as Peter I. Frazier [14].

3.5 Model Initialization

The GP Regression model uses the current data to estimate the objective function. During the initial phase, when there are very few data points, finding the global maximum of the acquisition functions can be challenging. This is because the mean and variance functions tend to flatten far from the evaluated points.

To address this, it is necessary to "initialize" the environment by performing some random evaluations within the feasible set of the objective function. This ensures that the Bayesian Optimization (BOpt) algorithm starts with sufficient information to operate effectively.

An efficient method for quasi-random sequence generation is the use of Sobol' Sequences. These sequences are utilized by Ax [20] during the initial phase of the optimization loop. In the following section, we will explain how Sobol' Sequences generate numbers.

3.5.1 Sobol' Sequences

Sobol' Sequences are a type of quasi-random low-discrepancy sequence used for generating points in a multidimensional space. They are particularly effective in the context of numerical integration and optimization due to their ability to uniformly cover the feasible set.

Unlike purely random sequences, Sobol' Sequences are designed to fill the space more uniformly. This property makes them highly suitable for the initial phase of Bayesian Optimization, where a good spread of initial points is crucial for building an accurate surrogate model.

Sobol' Sequences were originally designed to possess the property that, for a real integrable function f over a domain D, the generated sequence x_n ensures

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) = \int_D f(\tau)\, d\tau$$

to converge as fast as possible.

The generation of Sobol' sequences relies on a method that utilizes direction numbers to ensure quasi-randomness. Each dimension of the sequence is produced using a unique set of these direction numbers, which facilitates the even distribution of points across the multidimensional space.

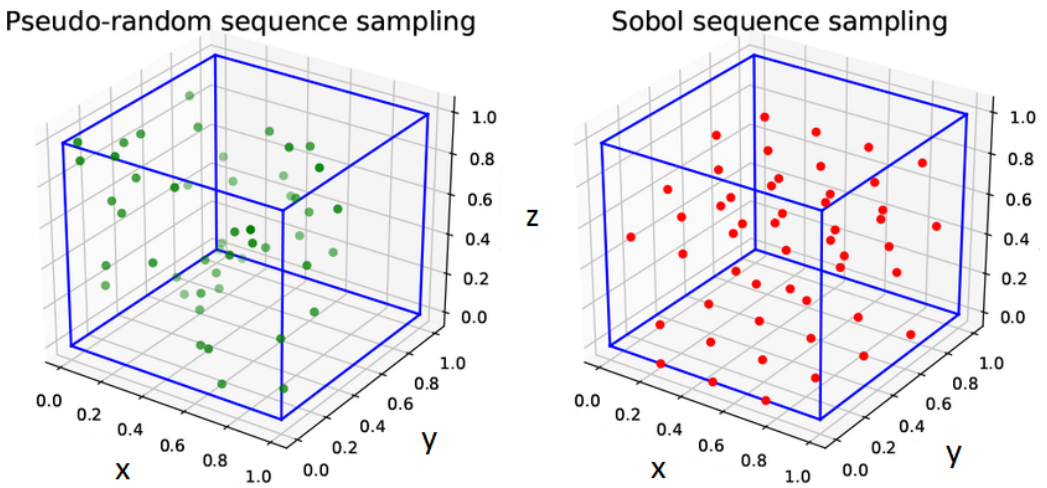


Figure 3.3. Pseudo-Random vs Sobol' generation

As illustrated in the image above, the property of Sobol' sequences to generate a set of quasi-evenly distributed numbers from direction numbers ensures quasi-randomness in computer experiments. This is a significant advantage over grid search methods, which not only fail to provide pseudo-randomness but also require a substantially larger number of points to adequately cover high-dimensional spaces.

In fact, a Sobol' sequence generated from a given set of direction numbers will always be the same, making it deterministic. The generation of this sequence is highly efficient, thanks to an algorithm proposed by Antonov and Saleev, which computes each subsequent point using only the previous one.

Both a theoretical study and practical implementation of a Sobol' sequence generator can be found in the work of Joe, Stephen, and Kuo, Frances Y. [18].

3.6 Integer Variables in Computer Experiments

As previously discussed, the underlying assumption for the objective function is that it is at least continuous. This assumption guarantees the existence of a maximum due to the validity of the following theorem:

Theorem 3.6.1 (Weierstrass)**.** *Let $f : A \to \mathbb{R}$ be a continuous function, where A is a compact subset of $\mathbb{R}^n$. Then f attains both a maximum value M and a minimum value m at some points within A.*

However, not all functions encountered in computer experiments will be continuous. Often, we need to work with ordered discrete sets, such as the number of neurons per layer in a neural network, or unordered discrete sets, like the choice of optimizer to use.

Fortunately, the feasible set is often straightforward to assess for membership, such as a hyper-rectangle or another simple shape. In such cases, as discussed earlier, this regularity condition allows us to round to the nearest integer value and proceed with the evaluations.

Given that Integer Programming is classified as NP-Hard, we can infer that these types of problems belong to that class or potentially even more challenging classes.

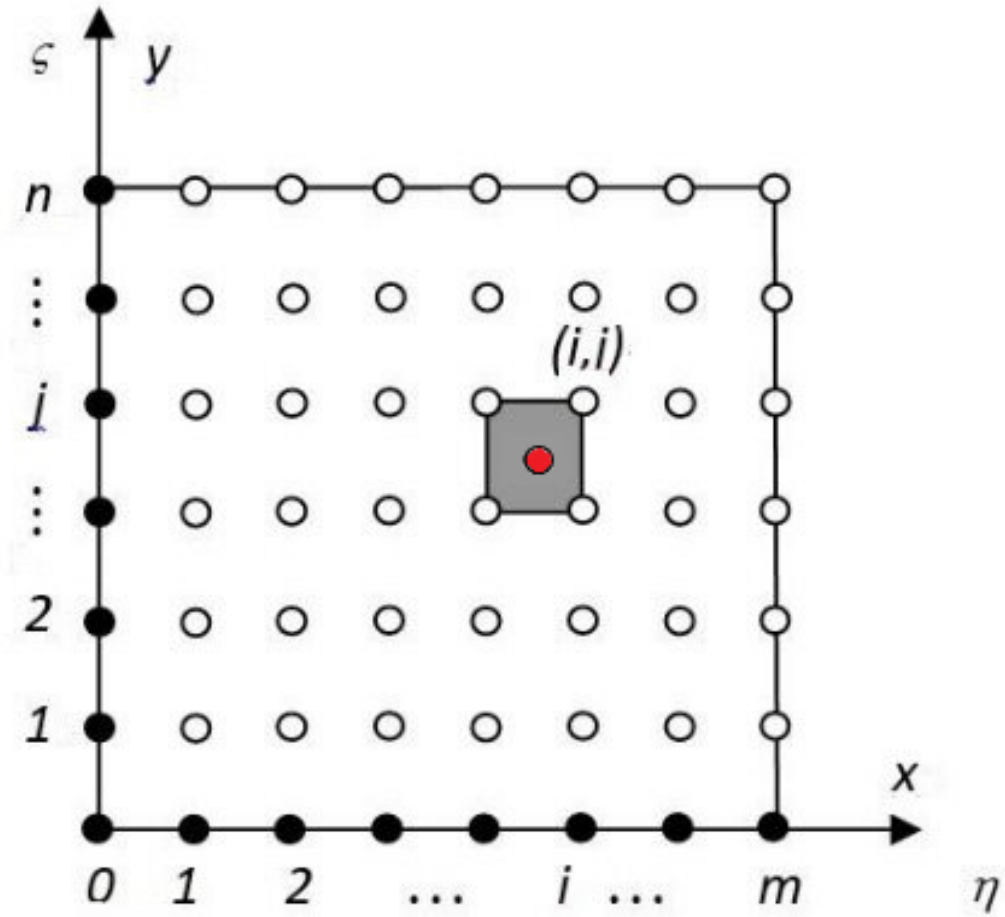


Figure 3.4. Here, the red point represents the maximum value under the relaxed problem, which will be rounded to the nearest integer.

3.7 Transformations to Smooth the Function

In the context of Bayesian Optimization, applying transformations to the objective function can be highly beneficial for achieving smoother behavior and enhancing the performance of optimization algorithms. Specifically, methods that rely on Gaussian Process (GP) Regression face challenges in constructing surrogate models for objective functions that exhibit high variations with small changes in inputs. These high variations can cause significant increases in variance, making the acquisition function difficult to use effectively. Two common transformations that address this issue are the logarithmic transformation and the reciprocal transformation.

3.7.1 Logarithmic Transformation

The logarithmic transformation is particularly useful when dealing with objective functions that span several orders of magnitude. By transforming the function $f(x)$ to $\log(f(x))$, we can reduce the effect of large variations and achieve a smoother, more tractable function.

$$g(x) = \log(f(x))$$

This transformation is especially effective when the original function values are strictly positive, as the logarithm is only defined for positive numbers. It can make multiplicative relationships linear and stabilize the variance.

3.7.2 Reciprocal Transformation

Another useful transformation is the reciprocal transformation, where the function $f(x)$ is transformed to $\frac{1}{f(x)}$. This transformation is particularly helpful when function values are large and need to be compressed into a smaller range. Although it can handle negative values, its smoothing effect is generally less pronounced than that of the logarithmic transformation.

$$h(x) = \frac{1}{f(x)}$$

It is especially beneficial in cases where the function values are neither zero nor approaching zero, as the reciprocal transformation is undefined at zero and tends to rapidly explode for small inputs.

3.7.3 Comparison and Application

The choice of transformation depends on the specific characteristics of the objective function and the nature of the problem being addressed. Applying these transformations can lead to more efficient and effective optimization processes by making the objective function more amenable to analysis and numerical methods. **In our experiment**, the output of the objective function is bounded between [0, 100], rendering both transformations unnecessary.

3.8 A Basic Pseudo-code for Bayesian Optimization

We now have all the necessary tools to construct the pseudo-code for the Bayesian Optimization loop to be used in our experiments, as shown below:

Algorithm 3: Bayesian Optimization Loop

```
Data: number of initial space-filling points n_0, total number of samples to
      produce N
Observe f at n_0 points according to a Sobol' sequence.
n ← n_0
while n ≤ N do
    Update the surrogate model using all available data.
    Let x_n be a maximizer of the chosen acquisition function.
    Compute the nearest integer x̂_n of x_n.
    Observe y_n = f(x̂_n).
    Increment n.
end
return The point with the largest f(x̂) value.
```

3.8.1 Multi-Objective Optimization and Outcome Constraints

In addition to the standard algorithm described above, the Ax implementation supports both multi-objective optimization and the application of outcome constraints on input or output variables. If these constraints are violated, the experiment is considered a failure.

For instance, outcome constraints might include achieving a specific level of precision or recall for a particular class in a multi-class classification problem, or optimizing the mean accuracy as the objective function.

3.9 BOTorch and Ax

In recent years, numerous platforms implementing Bayesian optimization algorithms have emerged. Among these, BOTorch and its wrapper, Ax, stand out as particularly robust frameworks for leveraging Bayesian optimization across a variety of applications. While other implementations built on TensorFlow and its Keras wrapper also exist, this work will focus exclusively on BOTorch and Ax.

3.9.1 Introduction to BOTorch

BOTorch is a library built on top of PyTorch, specifically designed for Bayesian optimization. It integrates seamlessly with PyTorch, offering high flexibility and efficiency in both modeling and optimization tasks. BOTorch's primary strength lies in its ability to handle complex, high-dimensional optimization problems with ease. This has been extensively discussed and analyzed by Andrew G. Wilson et al. [17], and it features prominently in numerous current peer-reviewed papers on Bayesian optimization.

BOTorch offers several key features that make it an attractive choice for researchers and practitioners. Notably, it implements all the features discussed in this

chapter and supports Quasi-Monte Carlo methods, custom optimizers for acquisition functions, and more, as detailed in its documentation [19]. These capabilities enable extensive customization of every aspect of the Bayesian Optimization Loop, establishing BOTorch as a versatile and powerful tool in the optimization toolkit.

3.9.2 Introduction to Ax

Ax , short for Adaptive Experimentation, was developed by the same authors of BOTorch to streamline the process of experimentation and optimization. It offers user-friendly APIs that range from simple interfaces to more detailed and customizable options, catering to a wide range of experimentation needs. These APIs include:

- **Loop API:** Intended for synchronous optimization loops where trials can be evaluated immediately. This API is suitable for the simplest use cases where running a single trial is fast, and only one trial runs at a time. Optimization can be executed in a single call, and experiment introspection is available once optimization is complete.
- **Service API:** Designed for parameter-tuning applications where trials might be evaluated in parallel, data is available asynchronously, or variables and constraints have a more complex structure, such as in hyperparameter or simulation optimization. It requires minimal knowledge of Ax data structures and integrates seamlessly with various schedulers. This API offers nearly complete hyperparameter optimization functionality without the need to understand the underlying architecture.
- **Developer API:** This API is for advanced users such as data scientists, machine learning engineers, and researchers who need extensive customization and introspection capabilities. It allows for detailed control over the entire experimentation process and is recommended for those optimizing complex structures such as A/B tests, customizing Ax, or leveraging advanced functionality. This API enables users to work with and edit the entire Bayesian optimization loop, providing unparalleled flexibility and control for new experiments.

In addition to these APIs, Ax also provides a **Scheduler**, which is an important use-case of the Developer API. The Scheduler allows for running a configurable, managed closed-loop optimization where trials are deployed and polled asynchronously without human intervention until the experiment is complete. It is suitable for experiments that need to interact with external systems to evaluate trials.

3.9.3 Additional Capabilities of Ax

Ax offers robust capabilities for data storage and management, supporting both local JSON and online SQL formats on a predetermined database. This allows for flexible and efficient data handling, ensuring that experiment data can be easily saved, retrieved, and shared, thereby facilitating seamless integration with other data processing pipelines.

Moreover, Ax supports man-in-the-loop optimization, incorporating human input into the optimization process. This capability is particularly useful in scenarios where expert knowledge or real-time decision-making is required, ensuring that the

optimization remains adaptive and responsive to complex, dynamic environments.

Additionally, Ax includes a range of advanced features such as Sobol' sequence generation, parallel evaluations, integer variables, discrete set ordered and unordered variables, outcome and parameter constraints, integrated metrics plotting, and more. These features significantly enhance its versatility, making it a powerful tool for a wide variety of optimization tasks.

3.9.4 Conclusion

The complete functionality and detailed implementation of these tools can be found in the BOTorch documentation [19] and the Ax documentation [20].

For our experiment, we will utilize the Service API, as it provides all the necessary features for hyperparameter optimization of our neural network.

Chapter 4

Experimental Outcomes

4.1 Rosenbrock Banana Function

In this section, we will discuss the results of the first experiment, which involves applying the Ax Service API to the Rosenbrock banana function. The Rosenbrock function is defined as follows:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

This function is non-convex and its gradient is given by:

$$\nabla f(x, y) = \begin{pmatrix} -2(a - x) - 4bx(y - x^2) \\ 2b(y - x^2) \end{pmatrix}$$

The Rosenbrock function has a global minimum at $(x^*, y^*) = (a, a^2)$, where $f(x^*, y^*) = 0$. It is well-known for being difficult to optimize using standard gradient descent methods. In fact, even using a random descent direction can be more efficient when employing Armijo's method. Moreover, this function can be efficiently optimized without any gradient information or local approximation models. In our experiment, we will employ this approach.

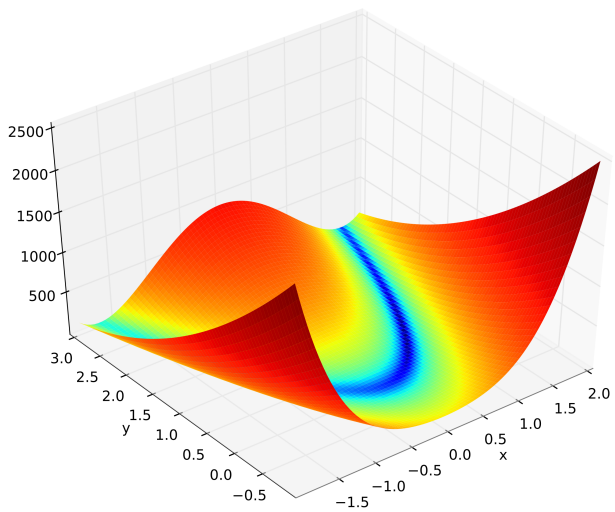


Figure 4.1. Rosenbrock function, with its banana shape.

We are now executing the Ax algorithm for the first time with a sequential setup, running 100 trials in which the first 5 are Sobol' generated.

$$\begin{cases} \min f(x,y) = (1-x)^2 + (y-x^2)^2 \\ x \in [-10, 10] \\ y \in [-10, 10] \end{cases} \tag{4.1}$$

By setting $a = b = 1$ and disregarding the noise assumption, the best result inferred by our algorithm occurs at the 91st trial with $(x, y) = (1.00292, 0.99658)$, truncated to the fifth decimal place.

We can now observe the results inferred by our model in the following contour plot, considering that we are treating this function as a black-box.

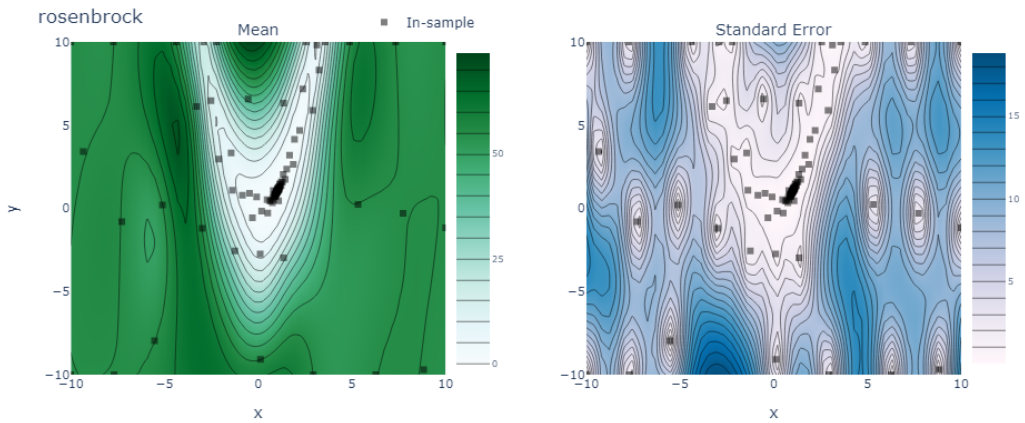


Figure 4.2. The inferred model of our Black-Box Function.

The characteristic banana shape is clearly recognizable. Notably, the Acquisition Function has successfully balanced both exploration and exploitation. This is evidenced by the numerous samples near the known optimum, as well as the more general samples distributed throughout the feasible domain. This success stems from Bayesian Optimization being a global optimization method rather than a local one. Lastly, let us examine the speed of convergence to the optimum in the next plot:

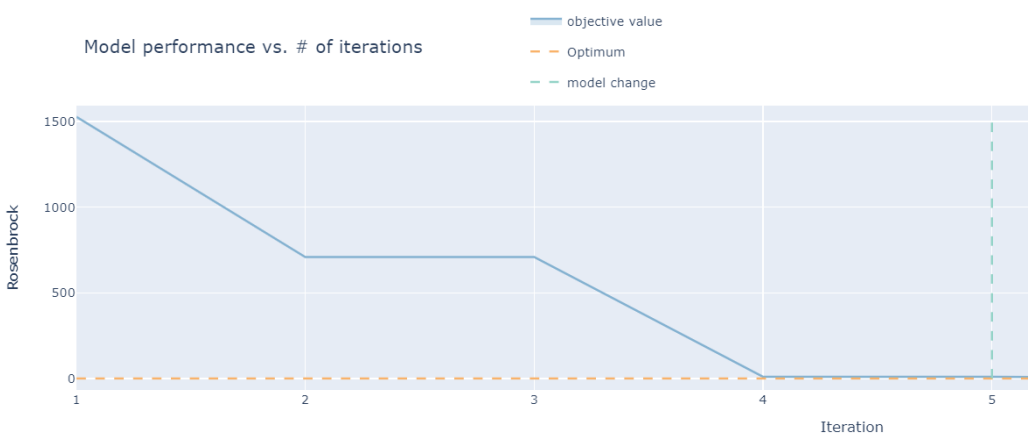


Figure 4.3. Model performance.

As we can see, after only the first few iterations, we already achieve a good proximity to our optimum. The results can potentially improve further, depending on the points chosen in the stochastic process. Different executions might perform better or worse, highlighting the variability inherent in stochastic processes. The algorithm is also effective in identifying a significant number of local minima without the risk of getting stuck, unlike gradient descent.

It is also well established that a smaller feasible set makes it easier to obtain good data. A useful approach could be to conduct a few initial trials, then narrow the domain around the latest best point. The model is then re-inferred, and the execution is restarted with this updated information, as showed in the next algorithm.

Algorithm 4: Ax Inference and Zoom

Data: a starting feasible set $\mathcal{L}$, a function $f(x)$ to minimize
$\Delta \leftarrow$ the inferred model of $f(x)$ on $\mathcal{L}$.
for $k = 0, 1, 2, ..., N$ **do**
  $\hat{x} \leftarrow$ the current best point found with Ax in Δ.
  $\mathcal{L} \leftarrow$ a smaller feasible set around $\hat{x}$.
  Δ is re-inferred using only the new currently feasible points in $\mathcal{L}$.
end
return $\hat{x}$

This technique can easily get stuck in local minima, but the restart feature allows for continuous updates to find the latest minimum. It then executes further optimization by building the model quickly, which prevents additional evaluations of the expensive function. This approach prioritizes the already known points, which are reused in our optimization loop. This method will be particularly useful later in our neural network training. If some points have already been evaluated before, we can skip the entire training process under a noise-less assumption, thereby saving resources for inferring the next point.

A noise-less assumption is very strong, but it can be reasonably applied in computer experiments where we can assume that a script running with the same parameters returns the same output every time. If this were not the case, evaluating the same point could be crucial to estimate the noise. Ax has a built-in function to infer noisy spaces.

Ultimately, one might think that Bayesian optimization is always preferable to other common methods like gradient descent. However, as the model becomes increasingly complex with each iteration, it becomes more difficult for Bayesian optimization algorithms, to find the optimum. In reality, even 500 iterations can take an impractically long time to infer the model compared to the actual function evaluation. This time can be easily overlooked when dealing with more computationally expensive functions.

Let us end this test analysis by showing what happens if we relax our domain to:

$$\begin{cases} x \in [-100, 100] \\ y \in [-100, 100] \end{cases} \tag{4.2}$$

In this case, our convergence is shown by this graph:

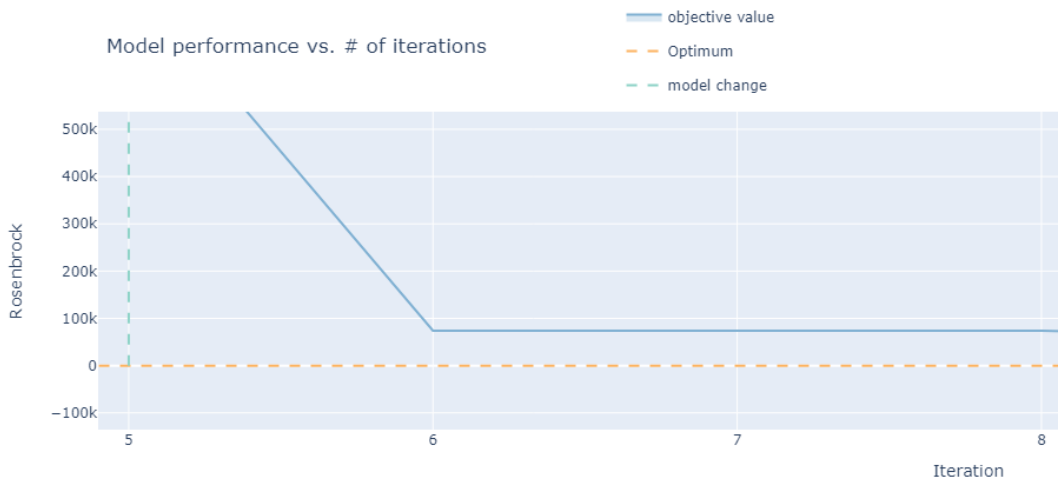


Figure 4.4. Convergence on a wider feasible set.

The best point found is at the 84th trial with $(x, y) = (0.81589, 0.74275)$, which is noticeably farther from the optimum compared to the previous results.

4.2 Neural Network Structure

In this section, we analyze the neural network implemented for our classification task, detailing its architecture.

4.2.1 Network Architecture

The proposed neural network comprises six layers with the following structure:

1. Convolution - Batch Normalization - ReLU
2. Convolution - Batch Normalization - ReLU
3. Max Pooling
4. Convolution - Batch Normalization - ReLU
5. Convolution - Batch Normalization - ReLU
6. Fully Connected (Linear)

4.2.2 Convolutional Layers

The convolutional layers utilize 5×5 kernels, with each layer consisting of multiple channels, each designed to detect specific features within the input data.

When configuring a convolutional layer, the following parameters must be specified:

- **Input Channels:** Number of channels in the input data.
- **Output Channels:** Number of channels produced by the convolution operation.
- **Kernel Size:** Dimensions of the convolution kernel.
- **Stride:** Number of pixels by which the filter moves across the input matrix.
- **Padding:** Number of pixels added to the input matrix around the border.

It is crucial to ensure that the output channels from one layer match the input channels of the subsequent layer to maintain dimensional consistency throughout the network. Additionally, padding is used to prevent data loss during the convolution operation by preserving the spatial dimensions of the input.

4.2.3 Batch Normalization Layer

The batch normalization layer normalizes the inputs to have zero mean and unit variance, thereby enhancing the network's accuracy.

4.2.4 ReLU Activation Function

We use the ReLU activation function in each layer. ReLU is popular for its simplicity and effectiveness in CNNs, as it helps prevent the vanishing gradient problem, as explained in 2.

4.2.5 Max Pooling Layer

The max pooling layer helps maintain the robustness of feature detection by ensuring that the network's ability to detect specific features is unaffected by the object's location within the image. It also reduces the spatial dimensions of the data, thereby decreasing the computational load and controlling overfitting by reducing the complexity of the network.

4.2.6 Fully Connected Layer

The final layer is a fully connected (linear) layer, which maps the learned high-level features to the desired output classes for the classification task. For the CIFAR-10 dataset, which includes ten classes of labels, the label with the highest score is predicted by the model. In the linear layer, the number of input features and output features (corresponding to the number of classes) must be specified.

4.2.7 Optimizer and Loss Function

For training the neural network, we used the Adam [7] optimizer, which is known for its efficiency and effectiveness in handling sparse gradients and non-stationary objectives. The loss function used is cross-entropy loss [11], which is appropriate for classification tasks as it measures the performance of a classification model whose output is a probability value between 0 and 1. Using cross-entropy loss, a softmax layer is not needed as the loss function inherently handles the computation of probabilities.

4.3 Hyperparameters Overview

In this section, we will review the set of parameters present in our model and determine which ones to use as variables in our Bayesian optimization loop.

4.3.1 Number of Layers

The number of convolutional and fully connected layers is an important hyperparameter. However, it is challenging to manage the dependent parameters because this variable influences the number of other variables in the function we are defining to optimize. Therefore, we will exclude this option in the near term.

An alternative implementation could involve using two loops: the first to select the number of layers, and the second to optimize the other parameters. This approach requires significantly more work and could be an extension of our project in the future.

4.3.2 Neurons per Layer

A common and intuitive choice is to determine the number of channels or neurons per layer. It is important to remember that the output channel of one layer is the input of the next.

Applying a multiplicative constant allows us to significantly scale the number of neurons in each trained model while keeping the variable as an integer (not a set), which facilitates the Ax configuration and enables the contour plot feature. For example, if n_i is the number of neurons in layer i, our equation would be $n_i = Kx_i$, where x_i is the input value from our Bayesian optimization loop and K is a constant, typically 12 or 16.

This constant K functions as a granularity factor and can be adjusted according to Ax Inference and Zoom (Alg 4). This adjustment is applied to the best point found, enabling us to search for a better local optimum. Additionally, K prevents Ax from selecting a variation of only 1-2 neurons per trial, which would cause minimal variation in overall accuracy and generate a locally flat utility function.

4.3.3 Kernel Size and Stride

Kernel size can also be considered a hyperparameter. Larger sizes may suffer more from noise but can capture more extensive features in the image. Stride and padding are calculated as a consequence of the chosen kernel size.

4.3.4 Activation Functions

The nonlinearity used in each layer can be varied, from ReLU to Sigmoid or Tanh, among others, as described in [9]. However, it is not recommended to change this frequently as it would be modeled as a set choice variable, and ReLU is a widely accepted standard in contemporary research.

4.3.5 Optimizer

The choice of the optimizer used in gradient descent is another important variable, along with its parameters that can be optimized. For example, the learning rate and, in the case of Adam [7], its forgetting factor β_1 and second momentum β_2 can also be included in the optimization process.

4.4 Chosen Variables

4.4.1 Curse of Dimensionality

Excluding the number of layers, a qualitative calculation suggests that we have more than 20 variables to consider! This is not advantageous because more dimensions require more trials to effectively optimize our function. Additionally, more trials are needed when using Sobol' sequences to properly initialize the model.

This phenomenon can be jokingly referred to as the *curse of dimensionality*, inspired by the challenges that classifiers face when the number of features increases.

4.4.2 Feature Engineering

Ideally, we would have an algorithm that could take all variables and find the optimal solution effortlessly. However, this is neither easy nor fast to achieve. We still face a feature engineering problem where we must select which variables are *important* and have a significant impact on our network's performance.

4.4.3 Utility Function

The last question we need to answer is, how can we measure our network's performance? More specifically, what performance metrics are necessary and in which categories? To address this, we must consider our previous discussions in 2, particularly which metrics can be used on our test set to evaluate the quality of our trials.

This depends on our goal. In our case, we have chosen to use the **average accuracy across all classes** as the utility function to maximize in our optimization loop.

Other approaches can also be implemented. For instance, we can set specific outcome constraints on precision or recall for certain classes that we consider more important than others. Alternatively, we could use the F1 score for each class in such cases or move to multi-objective optimization and use more than one metric to build our utility function, comparing them using the Pareto Frontier.

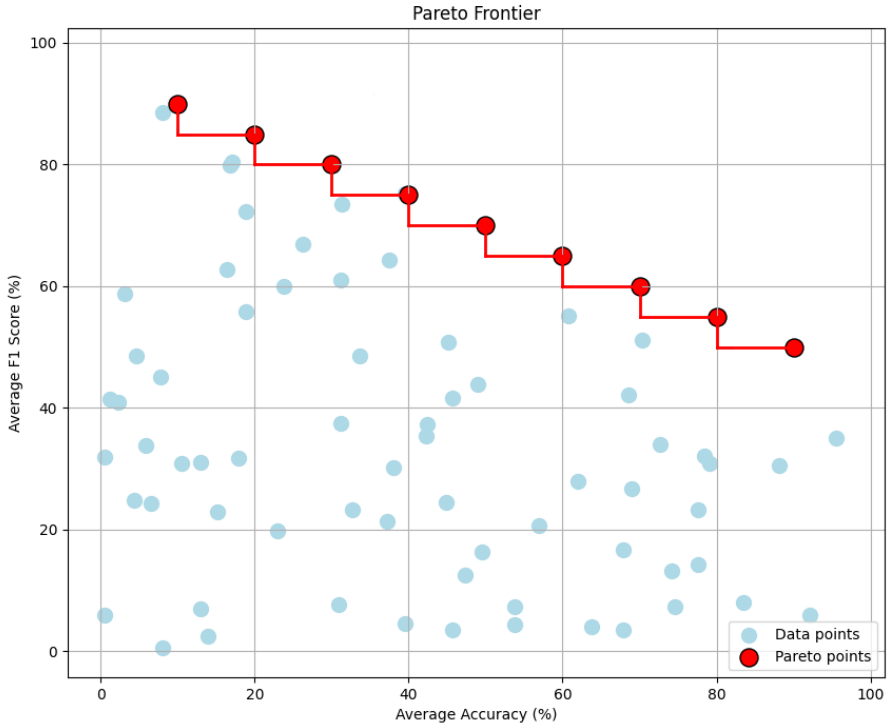


Figure 4.5. F1 Score vs Accuracy

4.4.4 Problem Formalization

In this project, we aim to maximize our *utility function* with respect to the number of neurons in each layer n_i, where l is the number of modifiable layers, and the learning rate α of the Adam Optimizer [7].

$$\begin{cases} \max U(n, \alpha) \\ n = K^T x \\ x_i \in [1, a_i] \\ n \in \mathbb{N}^l \\ x \in \mathbb{N}^l \\ K \in \mathbb{N}^l \\ \alpha \in \{10^{-2}, 10^{-3}, 10^{-4}\} \end{cases} \tag{4.3}$$

In this case, $K^T = (12, 12, ..., 12)$ and $\forall i, a_i = 16$. All variables are positive integers, and the learning rate is selected from a discrete set of logarithmic values. The utility function is bounded, specifically $U(n, \alpha) \in [0, 100] \subset \mathbb{R}$, so no logarithmic transformation or inverse transformation is needed.

4.5 First Discoveries

Our initial training involved manually training our network, resulting in an average accuracy of 73%, with $n^T = (12, 12, 24, 24)$ and $\alpha = 10^{-3}$.

In our first experiments, we observed that even during the Sobol' sequence generation, a better result was achieved at the 6th trial with $U_6(n, \alpha) = 79.73\%$. The maximum was found at the 22nd trial with $U_{22}(n, \alpha) = 80.85\%$. The algorithm was run for 50 trials.

4.5.1 Unexpected Event

Starting from the 22nd trial, the Bayesian optimization algorithm consistently decided to optimize the same point. This can be explained by the fact that we are dealing with an integer optimization problem. When relaxing to real numbers to find the maximum of our acquisition function, we repeatedly obtained a point near the sampled one, which was then truncated to the same point.

As a result, we evaluated the same point approximately 30 times, which is very inefficient for such an expensive function. To prevent this in future experiments, we introduced a check that prevents our neural network from starting training if the point has already been evaluated, returning the last known point instead.

This issue could likely be attributed to the bounds being too small. To address this, we relaxed the bounds in the next experiments, changing a_i from 8 to 16.

4.5.2 First Results

These are the trials performed and the results obtained. Note that each trial took approximately 25 minutes on an NVIDIA RTX 3060, leveraging CUDA for computation. Neglecting the Bayesian optimization computation, the total experiment time for 50 samples was around 21 hours.

# Trial	Method	n_1	n_2	n_3	n_4	α	Accuracy
1	Manual	12	12	24	24	10^{-3}	74.05
2	Sobol	60	48	36	84	10^{-2}	64.81
3	Sobol	24	96	84	24	10^{-2}	64.1
4	Sobol	36	36	36	84	10^{-3}	77.93
5	Sobol	72	24	12	12	10^{-3}	73.32
6	Sobol	60	60	60	24	10^{-3}	77.77
7	Sobol	72	72	60	84	10^{-2}	64.81
8	Sobol	84	36	96	36	10^{-4}	78.7
9	Sobol	60	84	24	48	10^{-4}	77.58
10	Sobol	36	96	72	96	10^{-4}	79.73
11	Sobol	96	12	12	60	10^{-3}	73.66
12	BoTorch	48	60	72	60	10^{-4}	79.1
13	BoTorch	48	72	60	72	10^{-4}	79.7
14	BoTorch	12	96	96	96	10^{-2}	64.66
15	BoTorch	96	12	12	12	10^{-2}	63.94
16	BoTorch	60	72	60	72	10^{-4}	79.19
17	BoTorch	36	72	60	72	10^{-4}	79.12
18	BoTorch	48	72	60	72	10^{-3}	78.77
19	BoTorch	48	72	72	72	10^{-3}	78.47
20	BoTorch	48	72	60	72	10^{-4}	78.95
21	BoTorch	48	72	72	72	10^{-4}	79.75
22	BoTorch	48	84	84	72	10^{-4}	80.65
23	BoTorch	48	96	96	60	10^{-4}	80.85
24	BoTorch	48	96	84	60	10^{-4}	80.38
25	BoTorch	48	84	96	72	10^{-4}	80.4
26	BoTorch	48	84	84	60	10^{-4}	80.03
27	BoTorch	60	96	96	72	10^{-4}	80.53
28	BoTorch	36	96	96	72	10^{-4}	79.86
29	BoTorch	60	96	96	48	10^{-4}	80.31
30	BoTorch	48	96	96	60	10^{-4}	80.85
31	BoTorch	48	96	96	60	10^{-4}	80.85
32	BoTorch	48	96	96	60	10^{-4}	80.85
33	BoTorch	48	96	96	60	10^{-4}	80.85
34	BoTorch	48	96	84	72	10^{-4}	80.12
35	BoTorch	60	96	96	60	10^{-4}	80.77
36	BoTorch	48	84	84	84	10^{-4}	80.27
37	BoTorch	12	12	96	12	10^{-2}	63.71
38	BoTorch	12	12	96	96	10^{-2}	63.51
39	BoTorch	48	60	60	60	10^{-3}	78.24
40	BoTorch	48	84	72	72	10^{-4}	79.35
41	BoTorch	48	60	60	96	10^{-4}	78.61
42	BoTorch	60	84	84	72	10^{-4}	80.57
43	BoTorch	48	96	96	60	10^{-4}	80.85
44	BoTorch	12	96	12	12	10^{-2}	63.18
45	BoTorch	48	96	96	60	10^{-4}	80.85
46	BoTorch	12	96	12	96	10^{-2}	63.42
47	BoTorch	48	96	96	60	10^{-4}	80.85
48	BoTorch	48	96	96	60	10^{-4}	80.85
49	BoTorch	48	96	96	60	10^{-4}	80.85
50	BoTorch	48	96	96	60	10^{-4}	80.85

Table 4.1. First Experiment results.

4.6 Improved Experimentation

In this section, we analyze our experiment results after tuning our Bayesian optimization loop. Some experiments attempted to treat the learning rate α as a continuous value on a logarithmic scale, but these attempts failed. This is because there is a tendency to use the smallest value for α. During each training trial, the loss function tends to oscillate with high values, preventing it from settling to the minimum correctly.

Another observation is that relaxing the bounds on the number of neurons significantly increased the network complexity, resulting in wider layers that further slowed down the training process. It could be beneficial to **add regularization techniques** such as Lasso or Ridge to prevent the network from overfitting, or add a cost penalty to the Utility Function.

We will also provide a useful comparison of the inferred model, separating the dimensions. Since our utility function $U(n, \alpha) : \mathbb{R}^5 \to \mathbb{R}$, it cannot be easily plotted in a standard graph.

The best results were achieved in the 46th trial, where $U_{46}(n, \alpha) = 82.21\%$ with the input $n^T = (192, 168, 168, 156)$ and $\alpha = 10^{-4}$. Ultimately, we improved our utility function by 8.47%, and we exceeded an 8% improvement even in the initial BoTorch trials. The experiment concluded in approximately 25 hours. All models are saved and stored, enabling testers to use additional metrics later to select the best model accordingly.

Another useful option is to save all experimental data and use it later to re-infer the model, starting another round of trials, as shown in the example below.

Algorithm 5: Ax Restart

Data: a feasible set $\mathcal{L}$, a function $f(x)$ to minimize, a set Σ of already done trials.
$\Delta \leftarrow$ the inferred model of $f(x)$ on $\mathcal{L}$ using Σ.
for $k = 0, 1, 2, ..., N$ **do**
  Do the usual bayesian optimization loop things...
end
$\hat{x} \leftarrow$ the current best point found with Ax.
return $\hat{x}$

# Trial	Method	n_1	n_2	n_3	n_4	α	Accuracy
1	Manual	12	12	24	24	10^{-3}	73.74
2	Sobol	120	96	60	156	10^{-2}	64.75
3	Sobol	84	72	96	72	10^{-2}	64.2
4	Sobol	192	48	132	60	10^{-2}	63.76
5	Sobol	84	12	24	96	10^{-2}	63.01
6	Sobol	156	156	84	144	10^{-4}	81.43
7	Sobol	120	168	132	120	10^{-3}	80.0
8	Sobol	12	84	144	96	10^{-4}	78.48
9	Sobol	96	180	168	168	10^{-3}	79.96
10	Sobol	108	36	72	120	10^{-3}	79.25
11	Sobol	60	12	96	36	10^{-3}	77.0
12	BoTorch	120	120	108	132	10^{-4}	81.44
13	BoTorch	96	156	60	120	10^{-4}	81.29
14	BoTorch	180	132	60	168	10^{-4}	81.68
15	BoTorch	180	156	168	132	10^{-4}	81.81
16	BoTorch	168	96	96	108	10^{-4}	81.6
17	BoTorch	168	120	120	108	10^{-4}	81.77
18	BoTorch	180	168	108	180	10^{-4}	81.83
19	BoTorch	180	96	168	108	10^{-4}	81.7
20	BoTorch	168	168	108	156	10^{-4}	81.59
21	BoTorch	180	168	84	144	10^{-4}	81.54
22	BoTorch	180	120	84	168	10^{-4}	81.54
23	BoTorch	192	180	132	168	10^{-4}	81.98
24	BoTorch	192	180	120	132	10^{-4}	81.84
25	BoTorch	168	180	108	168	10^{-4}	81.73
26	BoTorch	180	192	96	156	10^{-4}	81.58
27	BoTorch	192	192	72	132	10^{-4}	81.47
28	BoTorch	192	180	120	144	10^{-4}	81.87
29	BoTorch	168	108	132	144	10^{-4}	81.73
30	BoTorch	180	168	84	156	10^{-4}	81.51
31	BoTorch	168	108	84	144	10^{-4}	80.63
32	BoTorch	168	120	156	120	10^{-4}	81.71
33	BoTorch	168	132	132	168	10^{-4}	81.51
34	BoTorch	192	180	108	120	10^{-4}	81.73
35	BoTorch	192	144	108	168	10^{-4}	81.59
36	BoTorch	192	180	96	144	10^{-4}	81.5
37	BoTorch	180	144	168	144	10^{-4}	81.94
38	BoTorch	192	168	132	132	10^{-4}	81.75
39	BoTorch	168	156	144	156	10^{-4}	81.86
40	BoTorch	180	132	144	156	10^{-4}	81.77
41	BoTorch	192	132	108	108	10^{-4}	81.46
42	BoTorch	180	180	144	144	10^{-4}	81.83
43	BoTorch	168	120	168	144	10^{-4}	81.56
44	BoTorch	192	156	144	144	10^{-4}	82.02
45	BoTorch	180	156	144	144	10^{-4}	81.75
46	BoTorch	192	168	168	156	10^{-4}	81.76
47	BoTorch	192	132	168	144	10^{-4}	82.21
48	BoTorch	156	168	120	156	10^{-4}	81.72
49	BoTorch	12	192	192	12	10^{-2}	64.14
50	BoTorch	108	108	120	120	10^{-4}	81.35

Table 4.2. Last Experiment results.

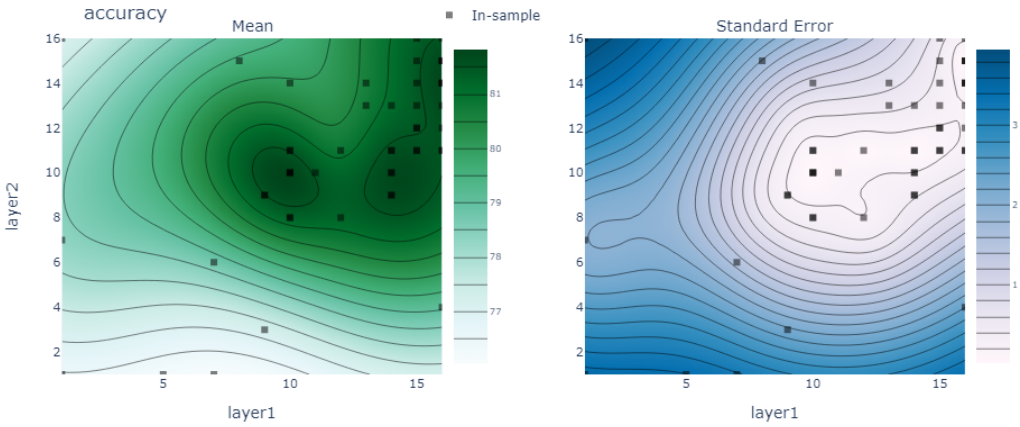


Figure 4.6. Contour Plot of x_1 and x_2

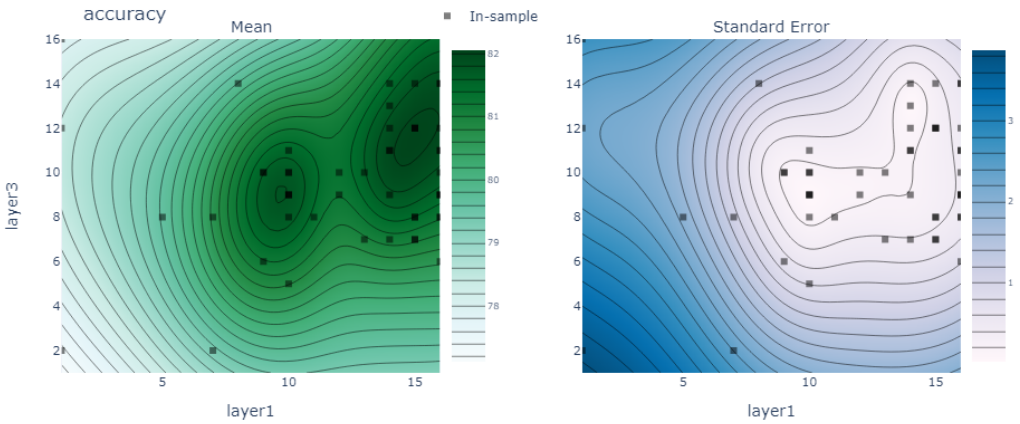


Figure 4.7. Contour Plot of x_1 and x_3

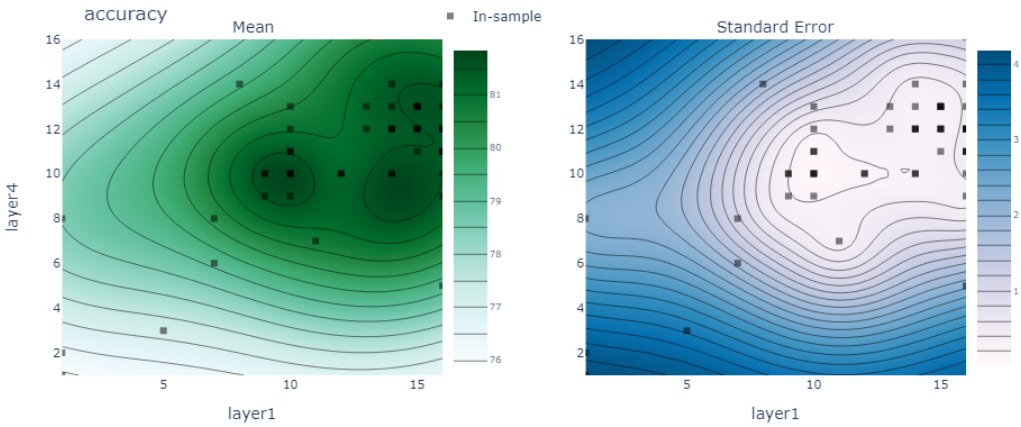


Figure 4.8. Contour Plot of x_1 and x_4

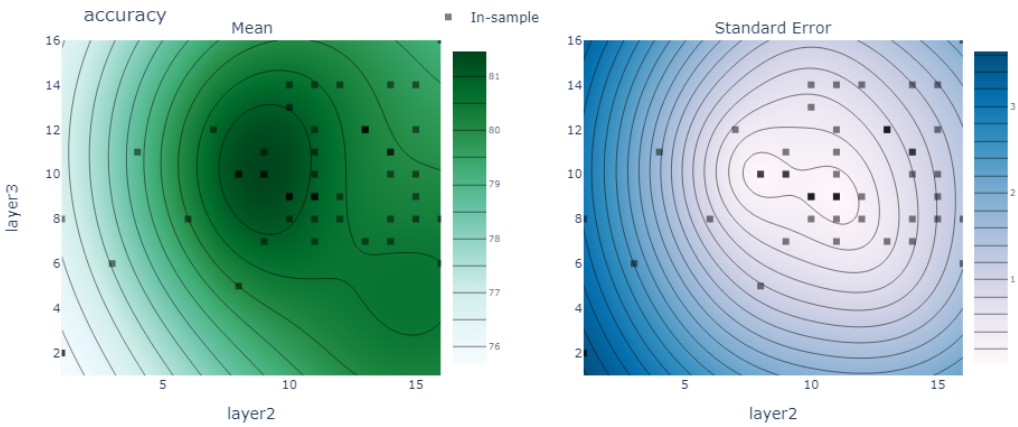


Figure 4.9. Contour Plot of x_2 and x_3

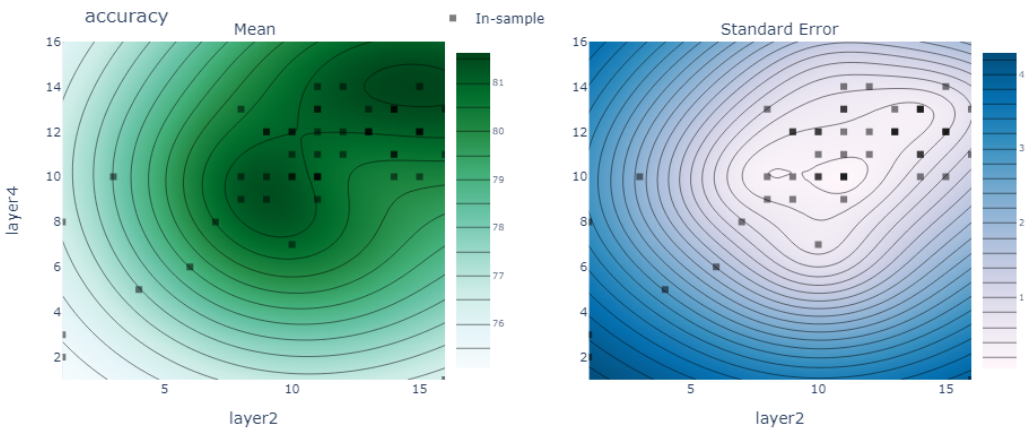


Figure 4.10. Contour Plot of x_2 and x_4

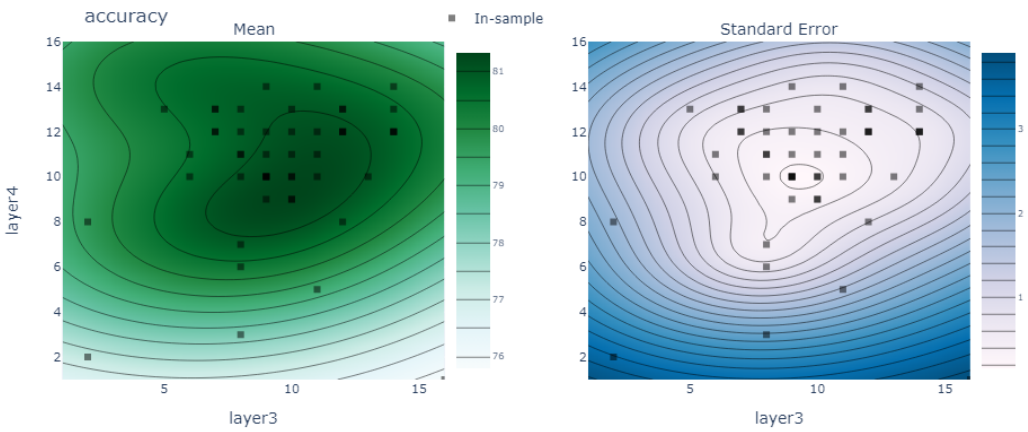


Figure 4.11. Contour Plot of x_3 and x_4

4.6.1 Plot Analysis

As we can see, all the plots of our inferred model are centered around the values we found. This suggests that we have likely identified the maximum of our function. To refine our results, we might consider increasing the density of our K values, allowing us to capture more granularity for our integers and potentially achieve a slightly better maximum.

The learning rate, configured in Ax as an ordered set, was not available for contour plots. Instead, it is represented by the brightness of the squares, where darker squares indicate a smaller learning rate.

In this last plot, it is evident that the convergence to a stationary point around $81 - 82\%$ is very rapid. By the 10th iteration, the model starts to infer after the Sobol' initialization.

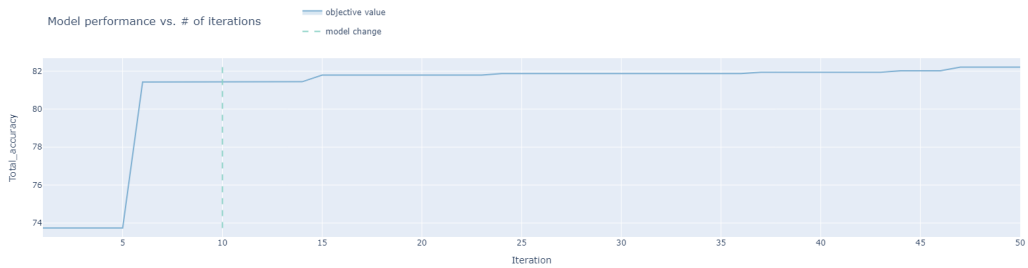


Figure 4.12. Convergence of our accuracy to the optimum.

Chapter 5

Conclusions

This project aimed to study Bayesian optimization as a derivative-free optimization method within the realm of mathematical programming.

Our studies have been applied to optimize a black-box, expensive-to-evaluate function, such as the accuracy of a neural network. The results were promising, as we automatically increased the accuracy by approximately 8%.

Given the obtained results, it is possible to assert that the project's purpose has been accomplished.

Numerous improvements can be made in the future. As repeatedly mentioned during our analysis, outcome constraints could be implemented, based on time and other metrics, as well as multi-objective optimization.

Entirely different fields of application can be considered, such as chemical reactions, where we could use a different acquisition function that allows for both parallelism and noisy environments.

Bibliography

[1] Massimo Roma, "*Appunti dalle lezioni di Ricerca Operativa*", 2022

[2] Narendra K. Karmarkar, "*A New Polynomial-Time Algorithm for Linear Programming*", AT&T Bell Laboratories, December 1984

[3] Raimund Seidel, "*The upper bound theorem for polytopes: an easy proof of its asymptotic version*", https://doi.org/10.1016/0925-7721(95)00013-Y, Computational Geometry 5 115-116, April 1995

[4] V. Klee and G.L. Minty, "*How good is the simplex algorithm?*" in O. Shisha (ed.) Inequalities III, Academic Press, 1972

[5] L. Grippo and M. Sciandrone, "*Introduction to Methods for Nonlinear Optimization*", https://hdl.handle.net/11573/1685119, Springer Nature, May 2023

[6] Giampaolo Liuzzi, "*Appunti di Analisi Matematica II*", December 2022

[7] D.P. Kingma and J.L. Ba, "*Adam: A Method for Stochastic Optimization*", https://doi.org/10.48550/arXiv.1412.6980, January 2017

[8] E.G. Birgin, J.L. Gardenghi, J.M. Martínez et al "*Worst-case evaluation complexity for unconstrained nonlinear optimization using high-order regularized models*", https://doi.org/10.1007/s10107-016-1065-8, Math. Program. 163, 359–368 Springer, August 2016

[9] S.R. Dubey, et al., "*Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*", https://doi.org/10.48550/arXiv.2109.14545, June 2022

[10] Razvan Pascanu, Tomas Mikolov and Yoshua Bengio, "*On the difficulty of training Recurrent Neural Networks*", https://doi.org/10.48550/arXiv.1211.5063, Feb 2013

[11] Juan Terven, et al., "*Loss Functions and Metrics in Deep Learning*", https://doi.org/10.48550/arXiv.2307.02694, September 2023

[12] PyTorch documentation, https://pytorch.org/docs/stable/index.html

[13] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton, "*The CIFAR-10 Dataset: Learning Multiple Layers of Features from Tiny Images*", https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[14] Peter I. Frazier "*A Tutorial on Bayesian Optimization*", https://doi.org/10.48550/arXiv.1807.02811, July 2018

[15] Görtler, et al., "*A Visual Exploration of Gaussian Processes*", Distill, https://distill.pub/2019/visual-exploration-gaussian-processes/ 2019.

[16] Donald R. Jones, Matthias Schonlau, and William J. Welch, "*Efficient Global Optimization of Expensive Black-Box Functions*", Journal of Global Optimization 13: 455–492, https://doi.org/10.1023/A:1008306431147, June 1998

[17] Andrew G. Wilson, et al., "*BOTORCH: A Framework for Efficient Monte-Carlo Bayesian Optimization*", Advances in Neural Information Processing Systems 33, https://doi.org/10.48550/arXiv.1910.06403, December 2020

[18] Joe, Stephen and Kuo, Frances Y., "*Remark on algorithm 659: Implementing Sobol's quasirandom sequence generator*", ACM Trans. Math. Softw., https://doi.org/10.1145/641876.641879, March 2003

[19] BoTorch documentation, https://botorch.org/docs/introduction

[20] Adaptive Experimentation Platform, https://ax.dev

Acknowledgements

Qui si conclude un percorso straordinario, che mai avrei immaginato potesse prendere questa direzione. Ora mi accingo a scrivere il capitolo più difficile di questa relazione.

Desidero ringraziare il mio relatore, il Prof. Giampaolo Liuzzi, per la passione che mi ha trasmesso durante i corsi di Analisi Matematica II e Ricerca Operativa. Al momento dell'iscrizione a questo corso di laurea, non avrei mai immaginato l'esistenza di questa disciplina. Lo ringrazio inoltre per tutto il supporto e il confronto critico che mi ha offerto nel corso di questo lavoro.

Dal profondo del cuore, desidero ringraziare i miei genitori, Mauro e Federica, per il grande supporto che mi hanno dato nei momenti alti e bassi della vita. Fin dalla nascita, la loro presenza e il loro esempio sono sempre stati per me fonte di ispirazione per affrontare le sfide più difficili.

Un ringraziamento speciale va anche alla parte statunitense della mia famiglia, a Kenneth ed Elaine Smith, così come a tutti i cari e affettuosi familiari sparsi per gli Stati Uniti.

Desidero esprimere la mia gratitudine a Soykat Amin, Federico Gerardi e Pietro Costanzi Fantini, compagni di vita, scuola, e ora di università, che mi hanno accompagnato fino a qui.

Un ringraziamento va anche a Federico Santavicca e Kristjan Jurij Tarantelli, conosciuti durante questo percorso, che mi hanno insegnato una prospettiva del mondo diversa e per nulla banale.

Un grande grazie a Cristian Di Iorio, Leonardo Miralli e Cristian Apostol, che mi hanno affiancato in questo cammino. Con loro ho condiviso momenti di noia, risate, scherzi e battaglie, nella spontaneità e spensieratezza che pochi al giorno d'oggi possiedono.

Infine, desidero ringraziare Claudio Totino e Francesco Andrea Placido, compagni di vita che, nonostante la distanza in questo mondo frenetico, sono sempre stati al mio fianco.