University of Bergen
Department of Informatics

Learning Acquisition Functions for Cost-aware Bayesian Optimization

Author: Audun Ljone Henriksen
Supervisors: Nello Blaser & Martin Møller Greve

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

August, 2023

Abstract

Bayesian Optimization is a powerful tool for optimizing expensive blackbox functions, with applications ranging from tuning hyperparameters in machine learning to optimizing portfolios in finance. In many real-life scenarios, such as the case of tuning hyperparamters in machine learning, the evaluation cost varies across the input space. However, few methods have been developed to work specifically with this type of function, and these just change already exisiting acquisition functions in a naive way.

This thesis presents a novel method for learning Acquisition Functions in Bayesian Optimization through the use of Reinforcement Learning. Using Reinforcement Learning allows the learned Acquisition Function to directly use information about evaluation cost to guide its decisions. Several experiments were conducted in order to test the performance of the proposed method, both when it comes to learning specific functions and when it comes to generalizing. The proposed approach demonstrated promising results, outperforming a wide variety of benchmarks in several different experiments. However, these results also highlight the need for further research in terms of the training process of the Acquisition Function, as well as the need for creating a more robust collection of experiments in order to test the proposed method on a wider range of optimization problems.

Acknowledgements

I would like to express gratitude to my supervisors, Nello Blaser and Martin Møller Greve. Nello for providing me with invaluable guidance and help throughout the writing of this thesis, and for always having an open door and minutes to spare to answer my questions and discuss ideas. Martin for inspiring the original goal of this thesis, aiding in improving the efficiency of solar panels. Sadly, I didn't find the time to test my method on this problem, but I am nonetheless thankful for his help, and for leading me down the path of cost-aware Bayesian Optimization.

I would also like to thank my mom, Britt, and my grandmother, Anna, for always believing in me and always supporting me.

I also want to thank my fellow students for all the great memories we have created together over the past two years, with special mentions going to Morten Blørstad for very valuable feedback on my thesis, as well as all the hours spent playing disc golf and discussing Reinforcement Learning.

Finally, I would like to thank my girlfriend, Ingrid, for her invaluable support and feedback, as well as her immense patience.

`Audun Ljone Henriksen`
Tuesday 1st August, 2023

Contents

Chapter 1

Introduction

In this chapter, I will first present the motivation behind the method that I aim to develop in this thesis. I then talk about the objective of the thesis, as well as the research questions I am attempting to answer. Lastly, I will present an outline for the rest of the thesis.

1.1 Motivation

Function optimization is a process of finding the inputs to a function that minimize or maximize the output. It is a set of problem-solving algorithms with very many use cases in all sorts of fields, such as optimizing portfolios in finance (Brochu et al. (2010)), minimizing downtime in logistics (Chhatre et al. (2022)), finding the best hyperparameters for a machine learning model, and so forth. These different function optimization methods have different advantages and disadvantages. Some can only be used if the function is differentiable and the gradients are known. Some methods are easier to parallelize than others, and others are more sample efficient.

Often, the problems that we face have inner workings we don't have access to, or we can't easily describe with a known set of equations. In these cases we refer to them as blackbox functions. Blackbox functions are functions whose internal structures are either unknown or inaccessible. Therefore, they are treated as simplified entities which simply accept inputs and generate outputs. The term is derived from the concept of a 'blackbox', which represents an object where the internal workings are hidden. Such situations commonly arise when data is sourced from simulations, physical experiments, or hyperparameter tuning. This inherent characteristic limits the number of function optimization algorithms that can be used, particularly excluding those that require access to the function's gradient.

Another common property of these functions is that they are costly to evaluate. This can either be costly in terms of financial resources, such as seismic surveys when drilling for oil, or costly in terms of time, such as hyperparameter tuning of a neural network where the performance of the network is only obtained after training for a significant amount of time. Working with expensive-to-evaluate functions means we have a multi-objective optimization scenario, where the goal is to both approach the optimum as closely as possible and to minimize the number of evaluations. Typically, there is a predetermined budget when dealing with this type of function, which restricts the total resources available for optimization. By predetermining this budget, we again turn this into a single-objective optimization scenario, where the goal is the find the optimum of the function by using the entire budget.

When we deal with functions that are both costly to evaluate and blackbox in nature, a technique called Bayesian Optimization is commonly used (Garnett (2023)). In short, Bayesian Optimization is a method that builds a probabilistic model of the function we want to optimize. Using this model, we can choose the next point we want to query by using an Aquisition Function, which is a function that guides our search using information from the probabilistic model. Based on the new information from the point we queried, we can update our probabilistic model, and iteratively get better and better guesses. Due to this, Bayesian Optimization is a sample-efficient method, meaning that it uses few function evaluations compared to other methods in order to approach an optimum. Since the biggest constraint when optimizing these expensive functions is cost, Bayesian Optimization is a common choice for this type of function.

In many cases, there might be a real-world cost constraint that limits how many resources we can spend on finding the maxima/minima of the function. A researcher might not have time to wait weeks to find the optimal hyperparameters for a neural network for an experiment. This means that how many times we can evaluate the function we are trying to optimize, depends on how long each evaluation takes. If we have 3 days, and each evaluation takes 6 hours, we can do a maximum of 12 function evaluations. Thus it is evident that the time each evaluation takes is important, as doing more function evaluations increases the odds of finding a good solution.

Input-dependent evaluation cost functions are functions where the cost of evaluation is not constant but rather depends on the input to the function. That is, the cost of evaluation is itself a function of the input to the function. If we can approximate or make an educated guess about the underlying cost function, we can use the relationship between the input and cost to increase the likelihood of approaching the function's optimum within our budget. This means that we can increase the number of samples we get within our budget by strategically sampling in parts of the input space associated with lower costs.

These three properties, namely being blackbox, expensive-to-evaluate, and having input-dependent evaluation cost, are the properties that define the functions that this thesis is concerned with. To illustrate these properties, consider a blackbox function representing the taste quality of bread given factors such as baking temperature, duration of baking, and rising time for the dough. Assume we have a budget of only 3 days to find the perfect recipe. The exact internal interactions affecting taste are not entirely known, making this a blackbox function.

Every combination of these parameters means baking a new loaf of bread and having it taste-tested by a panel, which is very time-consuming and thus makes this function expensive to evaluate. Additionally, the total time invested in each baking trial, which adds to the evaluation cost, depends on variables like baking duration and the dough's rising time, thus introducing an input-dependent evaluation cost.

In this thesis, I am going to propose a method for utilizing this cost dependency information in order to do more efficient function optimization. This will be done by making an Aquisition Function that takes cost information into account and is learned using Reinforcement Learning. I will compare this new Aquisition Function to a wide variety of benchmarks in order to assess its performance on the type of functions I am considering in this thesis. I also make a contribution in the form of developing a Reinforcement Learning environment that runs fully on GPU, providing a quick way of training the proposed method.

1.2 Objective

In this thesis, I will aim to investigate the implications of factoring in the cost aspect of the function evaluations on the performance of Bayesian Optimization by using Reinforcement Learning to learn an Aquisition Function. I seek to determine if exploiting evaluation cost information leads to increased performance of the Aquisition Function, both on specific function types, as well as in the general case.

More formally, I will attempt to answer the following research questions:

- Does the learned Aquisition Function utilize the evaluation cost aspect of the function evaluation in order to increase its performance compared to a learned Aquisition Function without access to temporal information?
- How does the performance of the learned Aquisition Function compare with other Aquisition Functions specifically developed for this type of problem?
- Is the learned Aquisition Function capable of generalizing beyond the function types it has been trained on?

1.3 Thesis Outline

The thesis is outlined in the following order. In Chapter 2, I will introduce the background theory needed in order to understand the method I have developed. Chapter 3 puts my work in context by talking about previous research in this field. In Chapter 4, I will go into detail on the method I have developed. The experiments I have designed, as well as the experimental setup, will be introduced in Chapter 5, along side the proposed Reinforcement Learning environment I have developed. I then present the results of these experiments in Chapter 6. Finally, I discuss the implications of the results and conclude my thesis in Chapter 7.

Chapter 2

Background

In this chapter, I will introduce the concepts and techniques that are needed in order to understand the research presented in this thesis. The chapter is divided into three sections: Function Optimization, Bayesian Optimization, and Reinforcement Learning.

The section on function optimization gives an overview of the aspects of optimizing functions, which is the process of finding the input to a function that returns the optimal value. I then introduce Bayesian Optimization (BO), the algorithm of choice in this thesis when it comes to function optimization. It is a method that is based on constructing a model of the underlying function we are attempting to optimize and using this model to make informed choices. It works especially well on the type of functions considered in this thesis, as introduced in Section 1.1. Lastly, I introduce Reinforcement Learning (RL), an area of machine learning that is especially well equipped to handle sequential decision-making problems, which function optimization often is.

2.1 Function Optimization

Function optimization is an important concept in many fields, including mathematics, statistics, and computer science. In function optimization, the goal is to find the input $\mathbf{x}$ to a function f that minimizes or maximizes the function output, depending on whether we are dealing with a minimization problem or a maximization problem. This section explores different types of function optimization algorithms and relevant constraints.

An optimization problem, as defined by Boyd and Vandenberghe (2004), is as follows:

$$\begin{aligned} \text{minimize} \quad & f_0(\mathbf{x}) \\ \text{subject to} \quad & f_i(\mathbf{x}) \leq b_i \quad \text{for} \quad i = 1, \ldots, n. \end{aligned}$$

In this representation:

- n is the number of dimensions in the objective function.
- $\mathbf{x} = (x_1, \ldots, x_n)$ is the variable we want to optimize.
- $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the function we want to minimize.
- $f_i : \mathbb{R}^n \to \mathbb{R}$ are constraints we want $\mathbf{x}$ to satisfy, where $i = 1, \ldots, n$.
- b_i are the upper limits for the constraints.

An optimal solution $\mathbf{x}^*$ is a vector that satisfies all constraints and for which $f_0(\mathbf{x}^*)$ is the largest possible value.[1] The different types of constraints will be discussed in the next subsection. The function we want to optimize, f_0, is often referred to as the objective function, which is the notation that will be used throughout this thesis.

There are two categories of optimization problems, depending on whether the input variable $\mathbf{x}$ is discrete or continuous. In this thesis, the focus will be on the latter, specifically the optimization problems with continuous input variables.

2.1.1 Constraints in Function Optimization

In an optimization problem, the objective function often has constraints on the input values. These constraints define a region in which the solution to the optimization problem is valid and thus limit the search space for our solution. There are generally two types of constraints we are concerned with in function optimization: equality constraints and inequality constraints (Boyd and Vandenberghe (2004)). Equality constraints demand that the input variables have an exact relationship. On the other hand, inequality constraints limit a variable, or a relationship between variables, to values within a specific range.

To illustrate these constraints, consider a company that has enough capital to produce up to 500 laptops or phones, assuming each cost the same to produce. The company wants

[1] Or smallest value, in the case of minimization problems

to determine how many of each product to manufacture in order to maximize profits. This problem can be represented as a function optimization problem with the function we want to optimize being $f(x, y)$, where $f(x, y)$ represents the total profit from selling x laptops and y phones. Because we want to produce exactly 500 phones or laptops, to maximize profits, and it's not physically possible to produce a negative number of either, the constraints, in this case, are $x + y = 500$, which is an equality constraint, and $x \geq 0$, $y \geq 0$, which are inequality constraints. These constraints are essential in representing real-world problems and deciding the strategy for optimization.

In this thesis, all the functions considered will have inequality constraints on the form $a_i \leq x_i \leq b_i$ where x_i is the ith input variable, and a_i and b_i are the corresponding bounds for the variable. However, the proposed method extends to functions with various types of constraints.

2.1.2 Algorithms for Continuous Optimization Problems

A myriad of techniques exist in the field of function optimization. The choice of a specific method should be based on the available information about the objective function, such as gradients, as well as the properties of the function itself, such as how noisy it is.

Gradient-based methods are advantageous when the gradient of the function can be evaluated at any point. Under such methods, at each timestep, the gradient of the function is calculated for a given input thus iteratively approaching a local extremum, usually a minimum. This is typically how the weights of a neural network are updated during training. Due to the deterministic nature of gradient-based methods, these algorithms are generally more sample-efficient than their stochastic counterparts. They are also scalable to thousands of dimensions or more, as seen with neural networks that have image data as input. However, as with all optimization techniques, they can potentially stagnate at local minima.

In cases where the function is non-differentiable or the internal structure of the function is unknown, such as in simulations of the physical world, alternative optimization strategies come into play. Techniques like Particle Swarm Optimization (Kennedy and Eberhart (1995)) or Genetic Algorithms (Holland (1975)) are often utilized. These are stochastic methods that investigate the function's landscape through a stochastic process. However, for functions that are expensive to evaluate, these methods are likely too sample-inefficient due to their stochasticity creating a high demand for function evaluations.

Surrogate models, mathematical approximations of the original function, become pivotal in dealing with such functions. They are probabilistic models that allow for efficient estimation of function values across the search space, using sparse amounts of data, and help identify promising areas for further exploration. This thesis particularly explores BO, a surrogate-based method, in subsequent section.

2.2 Bayesian Optimization

BO is a commonly used method when dealing with expensive-to-evaluate blackbox functions. It aims to find the optimal input $\mathbf{x}^*$ for a given blackbox function $f(x)$ with a limited number of evaluations and no knowledge of the inner workings of the system, making it suitable for the type of functions I will consider in this thesis. As it doesn't rely on randomness for sampling, it is an efficient method for exploring the input space. This efficiency makes it well-suited for optimizing expensive-to-evaluate blackbox functions, where it is a powerful, widely applicable tool (Shahriari et al. (2016)). In BO, our explicit goal is usually formulated as wanting to minimize the simple regret, which is the distance from the best-found solution to the global optimum of the function.

BO utilizes two critical components: a surrogate model and an Aquisition Function (AF), both of which will be explored in depth later in this section. The surrogate model is a mathematical approximation of the objective function. It's employed because it's less costly and easier to evaluate than the actual function, which is particularly valuable when optimizing expensive-to-evaluate blackbox functions. The AF uses information about the underlying function at a given point x_i, typically the predicted value in this point made by the surrogate model, and outputs the expected utility of sampling at that point. This utility measure assists in balancing the exploration of new regions and exploitation of known promising regions in the input space, leading to efficient optimization and reduced risk of getting stuck in local optima.

The foundation of BO is Bayes' Theorem, which underlies the process of iteratively updating our belief about the objective function. The process begins with a prior belief, embodied in our choice of surrogate model. As new data is collected, this belief is updated to a posterior via Bayes' Theorem. This posterior is then used to select a new point to sample from, progressively refining our approximation of the function.

BO, as depicted in Algorithm 1, uses a surrogate model to approximate the behavior of the objective function across the input space. An AF is then utilized in order to select a new point to sample. The AF ideally balances between regions promising optimal function values, i.e. exploiting current knowledge, and regions providing maximum information about the function, i.e. exploration of regions with high uncertainty in the estimates. With each iteration, a new data point is added, improving the accuracy of the surrogate model's predictions and, consequently, the quality of our function estimates.

Algorithm 1 Cost-aware BO (Based on Shahriari et al. (2016))

```
1: **Input:** resource budget $b$, initial data points $D$
2: Construct a surrogate model $m$ based on existing data $D$
3: **while** $b > 0$ **do**
4:     Identify the next query point $\mathbf{x_{n+1}}$ by maximizing the AF under $m$
5:     Estimate the expected cost $e$ for querying the objective function at $x_{n+1}$
6:     **if** $e \leq b$ **then**
7:         Query the objective function at $\mathbf{x_{n+1}}$ to retrieve new observation $y_{n+1}$ and actual cost $c$
8:         Deduct the cost $c$ from remaining budget $b$
9:         Append the new observation $(\mathbf{x_{n+1}}, y_{n+1})$ to dataset $D$
10:        Update surrogate model $m$ using the enhanced dataset $D$
11:    **else**
12:        **break**
13:    **end if**
14: **end while**
```

The central premise of BO is this iterative process shown in 1. The chosen AF decides the trade-off between exploration and exploitation. This allows the algorithm to efficiently seek out the function's optima while minimizing the number of function evaluations.

As mentioned in the introduction, the functions we are looking to optimize have three key components. BO already works well with two of them, which is expensive to evaluate blackbox functions. However, BO also allows us to take advantage of the third property, the input-dependent evaluation cost. This is possible by creating an AF, the part that guides our search, that directly utilizes the expected cost when guiding the search. For this reason, BO is the optimization algorithm of choice.

2.2.1 Surrogate Models

A surrogate model is a mathematical approximation of the objective function, created using sparse data from the objective function. Since the objective function is too expensive to directly optimize the objective function, we instead make an approximate model that is easily optimized. This gives us an idea of what the underlying function looks like, and thus it provides information that we use in order to sample the next data point. By carefully selecting a surrogate model that is easy to optimize in terms of computational cost, ideally by being differentiable, we effectively transform our optimization problem into an easier one by optimizing the approximate model instead of the objective function. We can then find the optima of this function by using more computationally efficient[2] optimization methods such as gradient descent or Particle Swarm Optimization.

Formally, a surrogate model the way that it is used in this thesis, can be defined as follows:

Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function that we want to model. A surrogate model $g : \mathbb{R}^n \to \mathbb{R}$ is then an attempt to approximate f such that $g(x) \approx f(x)$ for all $\mathbf{x}$ in the input space. The degree to which g matches f may vary depending on the data available and the method used to construct the model. The goal is to choose a g such that it is a good approximation of f as well as quick to evaluate.

The choice of surrogate model is how we encode our prior belief of the underlying function in BO. Therefore, the selection of a suitable surrogate model is crucial. Many methods exist to estimate this function, each with its strengths and weaknesses. Some of these methods, such as random forests or neural networks, natively only provide us with an estimate of the function value at a given point. Other methods, such as Gaussian Processes (GPs) or Parzen-Tree Estimators, give both an estimate of the function value at a given point as well as the uncertainty in that estimate. This uncertainty aids BO by balancing exploration and exploitation.

In this thesis, GPs are the surrogate models of choice, primarily due to their inherent properties, which makes them well-suited for BO. GPs allows for the easy computation of gradients with respect to the input variable, thus enabling the application of gradient-based optimization techniques to find the optima of the GP. Another major advantage is the high expressiveness of GPs, which can be harnessed by selecting an appropriate kernel function. This permits the modeling of a wide variety of behaviors, including periodic patterns, which enables the incorporation of any prior knowledge about the objective function directly into the model. Lastly, the robustness of GPs in handling noisy inputs makes them suitable for many real-life problems, although in this thesis we are only concerned with functions that have no noise in their evaluations.

[2]BO is fairly computationally inefficient due to the need to fit an expensive surrogate model to the available data

Gaussian Processes

A GP is a stochastic process used to model functions and associated uncertainties, essentially extending the Gaussian probability distribution to infinite-dimensional spaces (Rasmussen and Williams (2006)). This representation captures distributions over functions, rather than random variables. A GP consists of a collection of random variables, any finite number of which have a joint Gaussian distribution. A GP is defined by a mean function $m(x)$ and a covariance function $k(x, x')$, or kernel. It typically assumes a constant mean function, with the kernel being the part that captures the properties of the function space.

Formally, a GP is denoted as:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')).$$

Here, $f(x)$ represents our guess for what the underlying function is. For each input value x it outputs a random variable. That is, it provides not a single number, but rather a distribution. As the name Gaussian Process implies, the output is the normal distribution. Thus the output of $f(x)$ for a single input value $\mathbf{x}$ is the mean μ and the standard deviation σ of this distribution, with the mean representing our best guess and the standard deviation representing the uncertainty in that guess.

In order to utilize a GP, we initially choose an appropriate kernel function. We then tune the parameters of the kernel by minimizing their log-likelihood, thereby tuning the kernel to capture some inherent properties of the underlying function. Predictions for new inputs are made by evaluating the kernel function between these inputs and existing data, and conditioning on the observed data. This provides the posterior predictive distribution, yielding both the expected mean and its associated uncertainty at the new location.

Despite the widespread use of GPs in BO, they are not without limitations. One prominent drawback is their performance deterioration when dealing with high-dimensional functions. This degradation is primarily due to the curse of dimensionality (Eriksson and Jankowiak (2021)). Another significant limitation is their computational scalability. Specifically, GPs have a cubic time complexity for fitting, represented as $O(n^3)$, where n is the number of data points, which makes them computationally intensive for large datasets (Belyaev et al. (2014)).

Kernels

Kernels are a critical component of GPs, serving as a crucial function that determines the correlation or similarity between two points in the input space, based on solely on their coordinates. This correlation is typically dependent on distance, with closer points typically having a higher correlation. Kernels can be combined (typically through addition or multiplication) to create more complex kernels to suit specific needs, such as creating a quadratic kernel by multiplying two linear kernels (Duvenaud (2014)).

Many kernels have been developed throughout the years, with distinct properties for different applications. These kernels have their own parameters, typically between two to three, and are usually learned from the data by minimizing the marginal log-likelihood. Domain knowledge plays a pivotal role in kernel selection, allowing for educated guesses about the underlying function and aiding in the choice of suitable kernels for the type of function to be optimized.

The Radial Basis Function (RBF) kernel, also known as the Squared Exponential Kernel, is a commonly used kernel. It is commonly considered the default kernel for GPs (Duvenaud (2014)). The RBF kernel is defined as follows:

$$k(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right). \tag{2.1}$$

Here, ℓ is the length scale parameter that determines the function's smoothness, and σ represents the noise variance of the kernel. The RBF kernel is characterized by these two hyperparameters. Usually, the length scale parameter is learned from the data, while the noise scale can be set directly if the noise level in the observations is known.

In general, the length scale ℓ controls the extrapolation distance from data points. That is, it describes how far away from any data point we can move before that data point no longer has any influence on the predicted value. Meanwhile, σ represents the data's noise level.

Another interesting kernel is the spectral mixture kernel (Wilson and Adams (2013)). It allows us to capture a wider range of patterns in the data, especially when modeling mixtures of different functions, and is very powerful if you know the underlying function is suitable for such a kernel.

The spectral mixture kernel is defined as:

$$k(x, x') = \sum_{q=1}^{Q} w_q \exp\left(-2\pi^2 (x - x')^2 v_q^2\right) \cos\left(2\pi (x - x') \mu_q\right) \tag{2.2}$$

In this definition, w_q, v_q, and μ_q denote the weight, variance, and mean of the q^{th} mixture component. The total number of mixture components is denoted by Q. Typically, Q is chosen by the domain expert, whereas w_q, v_q, and μ_q are learned from the data.

2.2.2 Aquisition Functions

An AF, denoted $\alpha(s_i)$, where s_i represents information about a candidate point x_i in the input space, measures the utility or desirability of sampling at point x_i. The purpose of an AF is to guide the sampling process in the optimization by assigning interesting points a higher score. Interesting is in the sense that they either lead to information gain or a better solution.

A good AF should possess three main qualities. First, it should be easy to optimize as we are primarily interested in the optima of the AF. Second, it should exhibit a strong correlation between its output and the promise of a point, where a 'promising' point strikes a good balance between high expected improvement and appropriate exploration-exploitation trade-off. Lastly, a good AF should facilitate a robust balance between exploration and exploitation, avoiding confinement in local optima.

As mentioned earlier, AFs typically consider both the mean (μ) and standard deviation (σ) that the surrogate model outputs for that point. They are commonly statistical in nature, making them relatively straightforward to interpret. One very commonly used AF is Expected Improvement (EI).

EI is defined as follows:

$$EI(x) = (\mu(\mathbf{x}) - f(\mathbf{x}^+) - \varepsilon)\Phi(Z) + \sigma(\mathbf{x})\phi(Z),$$

where $Z = (\mu(\mathbf{x}) - f(\mathbf{x}) - \varepsilon)/\sigma(\mathbf{x})$[3], $\mathbf{x}^+$ is the current best input point, Φ and ϕ denote the CDF and PDF of the standard normal distribution, respectively. The output of the EI function is easily interpretable, that is, for a given point x_i, it simply measures how much we expected to improve over the current best-found solution, by sampling x_i.

[3] Z represents the standardized improvement over the current best point $\mathbf{x}^+$, and thus controls the trade-off between exploitation and exploration. The ε parameter adjusts the balance between exploration and exploitation.

2.2.3 Visualization of Bayesian Optimization

The upper plot in Figure 2.1 shows an example of a GP at optimization step t. By using the observed data, the GP predicts the underlying function $f(x)$. The lower plot shows that the EI for the above GP has its maxima at $x = 8.89$.

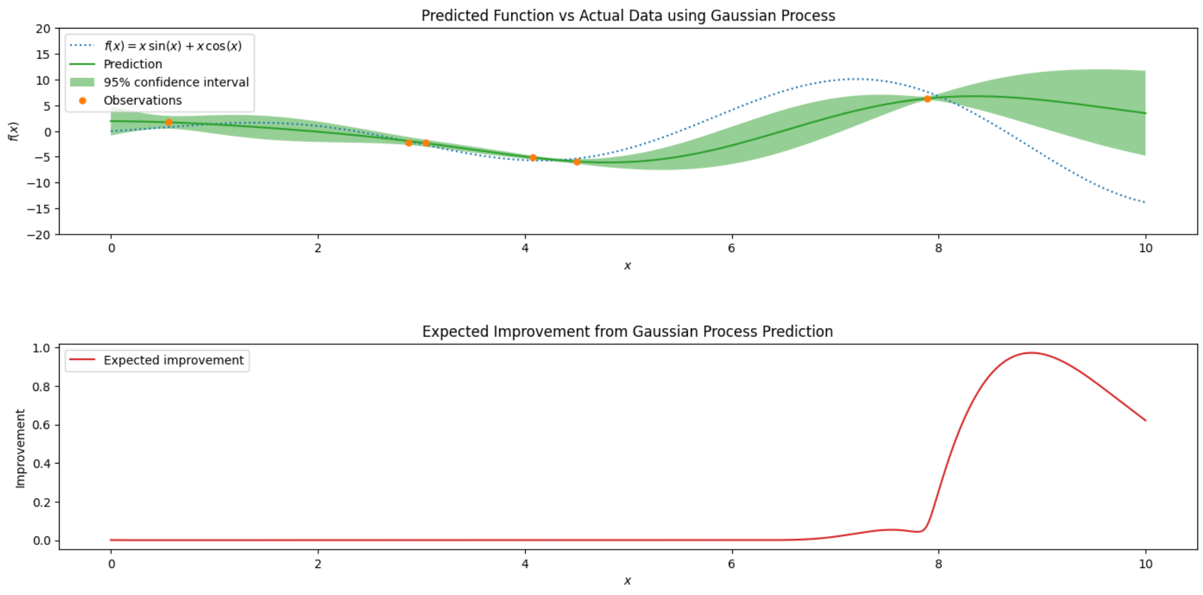

Figure 2.1: Top: Predicted function by Gaussian Process at timestep t. Bottom: Corresponding Expected Improvement, with its maximum at $x = 8.89$.

By sampling the underlying function $f(x)$ where the EI has its maxima, we obtain a new data point. We then append this data point to our existing observations and use the expanded dataset to fit the new GP seen in the upper plot of Figure 2.2.

As seen in Figure 2.2, the maxima of the EI has shifted to $x = 7.35$. This is closer to the actual maxima of $f(x)$, and by sampling this point, we obtain a new best-found solution. This process continues until we have exhausted our evaluation budget.

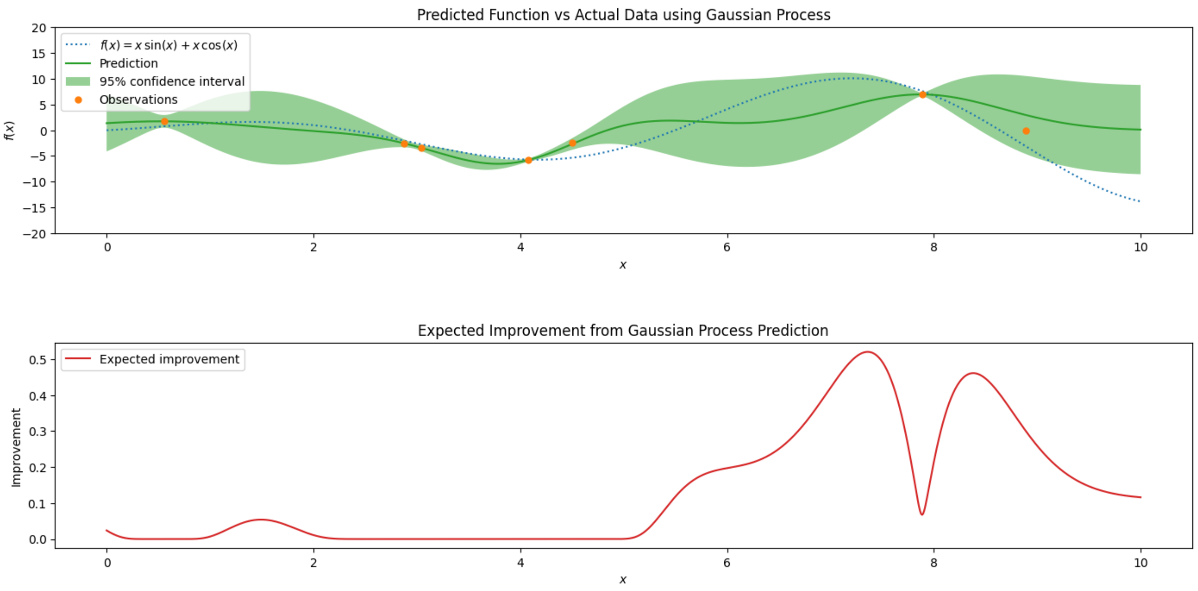

Figure 2.2: Top: Predicted function by Gaussian Process at timestep t + 1, after updating with new observation. Bottom: Corresponding Expected Improvement, now maximized at $x = 7.35$.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is one of three main types of machine learning, alongside supervised and unsupervised learning (Sutton and Barto (2018)). In RL, we train an agent to learn how to act in an environment by taking actions. A RL setup involves an agent interacting with an environment in a sequence of discrete timesteps. At each timestep t, the agent receives a state s_t from the environment and, based on this state, chooses an action a_t. The environment processes this action and gives the agent a new state s_{t+1} as well as a reward r_t. The goal of the agent is then to maximize the expected return $E[G]$, which is the sum of the cumulative rewards.

2.3.1 State

While the term 'state' can be used to refer to the environment state, which is all the information describing the environment at a specific point in time, it usually refers to the state representation in the context of RL. The state representation is what the agent actually observes from the environment and is what will be referred to as a state from here on. While these two can be the same in the case of a fully observable state, the state is often not fully observable. For instance, in a game of poker, the environment state includes the hands of all players and the communal cards. However, an individual player does not have access to this complete state. Each player can only observe their own cards and the communal cards. This means that the agent's state representation is only a partial view of the environment state. Other players' cards retain hidden information, and the state is thus not fully observable.

Another key concept in a state representation, which is crucial for many theoretical guarantees in RL, is the notion of a Markov state. A Markov state s_t is a state where the history of previous actions is irrelevant to the future evolution of the environment. In other words, the sequence of actions $a_0, a_1, ...a_{t-1}$ taken to arrive at s_t doesn't impact the system. A state that satisfies this property is said to satisfy the Markov property.

To explain a Markov state through an example, consider a chess game where the state representation is the position of all the pieces on the board. An important rule in chess is that if the same game state is reached 3 times, the game can be ended as a draw. [4] If we only know the current position of the pieces, we do not have a Markov state representation. This is because the current probability of the game ending in a draw given an action is not only dependent on the current state representation but also on the history of actions. A state representation that satisfies the Markov property would also include information about previous board configurations. [5]. Note that we often use the term state space to refer to the collection of all possible states.

2.3.2 Action

An action in RL is what the agent uses to interact with the environment. An action is either continuous, where an action takes on some value in a range, such as the angle of a joint in a robot, or it is taken from a discrete set of possible actions, such as in the case of chess.

An action can also consist of multiple components, and each component can be discrete or continuous. To expand on the example earlier, there are usually multiple joints to consider when controlling a robot, and an angle needs to be set for each one. In such a case, one action is represented by more than one component.

[4]Per the rules of Fide (FIDE (2021)), if the same game state has been reached 3 times, the player whose turn it currently is can force a draw.

[5]There are numerous other things in Chess that need to be included in order to truly satisfy the Markov property, namely castling rights and en passant.

In the context of BO, an action, or a sample location, is given by coordinates for a point. Since an n-dimensional point is given by n components, each one being a continuous value, we have a multi-action continuous action space.

2.3.3 Environment

An environment in the context of RL is the system that the agent interacts with. It is typically modeled as a Markov Decision Process (MDP). An MDP is a mathematical framework for modeling sequential decision-making processes (Sutton and Barto (2018)). It is defined as a 4-tuple (S, A, P, R), where

- S is the state space, as defined in 2.3.1
- A is the action space, that is, the collection of all possible actions. Commonly the set of available actions is independent of the current state. Still, it doesn't have to be.
- P is the function $P_a(s, s^{'})$ defining the probability of transitioning from state s to state $s^{'}$ due to action a. Formally, it is defined as such: $Pr(s_{t+1} = s^{'} | s_t = s, a_t = a)$. P defines the dynamics of the environment.
- R is the reward function. The reward function $R_a(s, s')$ defines the reward given for transitioning from state s to state $s^{'}$ due to action a.

This 4-tuple contains everything we need to fully define the environment. Note that an environment can either be episodic or continuous, with episodic meaning that a terminal state T is reached in finite time, whereas continuous means that the environment will never reach a terminal state.

2.3.4 Policy

In RL, a policy is a strategy that the agent uses in order to choose the next action based on the current state. It is thus a mapping from a state s_t to the corresponding action a_t. Formally, it is a function $\pi : S \rightarrow A$, where S is the state space and A is the action space. A policy can either be deterministic, where $\pi(s)$ maps directly to an action, or probabilistic, where $\pi(a|s)$ maps to a probability distribution over actions. In reality, probabilistic policies are more commonly used during training to allow for exploration of the state space in the environment.

Policy Types

There are two main types of policies, value-based, such as Q-learning, and policy-based, such as Proximal Policy Optimization (PPO) [6], which we will go more in-depth on later. These methods have their advantages and disadvantages, such as Q-learning not inherently supporting continuous actions or probabilistic policies.

Value-based methods learn an action-value function, a function $Q(s, a)$ [7] that maps from state s and action a to expected return G. A policy is then derived from this function, commonly by taking the action that maximizes the expected return through $\pi(s) = \arg\max_a Q(s, a)$. Exploration is then added by randomly sampling an action instead of following the policy. How often the action is sampled rather than derived from the action-value function is dependent on a hyperparameter ϵ.

Policy-based methods learn a parameterized policy $\pi_\theta(a|s)$, a function that maps from state s to a probability distribution over actions a. This policy is parameterized by θ and assigns to each action a a probability $\pi_\theta(a|s)$ of being taken when in state s. During training, an action is sampled from this probability distribution: $a \sim \pi_\theta(\cdot|s)$. Thus, exploration is inherently built into the policy, and it can be state-dependent.

Representing the Policy

Both value-based methods and policy-based methods provide a mapping from a state to an output - a scalar for the action-value function in value-based methods, or a vector representing an action distribution in policy-based methods. This can either be done with a tabular approach or with function approximation. Both of these methods are viable in this thesis, each with its strength and weaknesses.

A tabular approach uses a table that directly maps from any state to a scalar or vector. During training it allows us to independently update the value associated with any given state, without affecting the value for other states. This allows them to potentially have guarantees of converging to the optimal policy. They also tend to require significantly less compute compared to function approximation. However, tabular methods are

[6]Note that PPO is actually a mixture between value-based and policy-based, as the critic part of the method is value-based. However, as the policy is directly learned, I choose to refer to it as policy-based in this thesis.

[7]If the dynamics of the system is known, that is, the transition probability P, a function $V(s)$ can be learned instead.

only viable when the state space is small enough to allow sufficient visits to each state for value estimation. Due to their simplicity, tabular methods typically involve fewer hyperparameters.

For large or continuous state spaces, function approximation methods are more suitable. Under the assumption that similar states should have similar optimal actions, these methods can generalize across states and extrapolate to unseen states. This method allows us to extrapolate, making reasonable predictions about unseen states based on learned patterns. Because updating one state's value influences other states' predictions, function approximation methods may require fewer updates to reach an acceptable policy, depending on the problem at hand. However, these methods can suffer from instability and lack of convergence guarantees, which is a trade-off for their ability to handle larger state spaces and their potential for faster learning. Many methods have been developed to stabilize these methods, but they are often complex or computationally expensive.

Another important difference between tabular methods and function approximation methods, specifically if using neural networks, is that only the latter can be optimized with gradient-based methods. Since this thesis is concerned with making a policy that acts as an AF, ease of optimization of the final policy is of great importance. Since gradient-based methods are commonly used for the purpose of finding the optima of the AF, having access to these is an advantage for the function approximation methods. Note that while gradients are not available for tabular methods, it is still possible to optimize them using other optimization algorithms, such as Particle Swarm Optimization.

2.3.5 Reward

The reward function, $R_a(s, s^{'})$, determines the reward given for transitioning from one state s to another state s' due to a particular action a. The design of an effective reward function is critical in RL, as it is used to steer the learning process. A well-crafted reward function rewards the agent for actions that align with the intended goal and penalizes actions that deviate from it. If the reward function is too sparse, meaning that rewards are infrequent, the learning signal can become too weak and be drowned out by noise, leading to minimal learning.

Take the game of chess as an example. The ultimate goal is to capture the opponent's king before they capture ours. If we assign a positive reward for capturing the enemy king, a negative reward for losing our king, and zero otherwise, the reward function aligns with our goal perfectly. However, since only one move per game will yield a reward, learning can become slow and unstable.

One way to address this is to reward the agent for capturing the opponent's pieces and penalize it for losing its own pieces. This not only makes the reward function denser but also aligns with the general chess strategy of maintaining more pieces than your opponent to increase the odds of winning. However, this solution poses another problem. If the reward for capturing enemy pieces is set too high relative to capturing the enemy king, we could inadvertently create a policy that prioritizes capturing all other pieces before attempting to capture the king, as doing so would end the game and eliminate the potential for further reward. Therefore, carefully crafting a reward function that encourages the desired behavior is key to learning a useful policy in RL.

2.3.6 Proximal Policy Optimization

PPO (Schulman et al. (2017)) is an Actor-Critic learning algorithm for RL that was developed by OpenAI. It has been used to solve numerous high-complexity problems, including natural language processing in its use in ChatGPT (Ouyang et al. (2022)) and achieving human-level performance in complex cooperative games such as Dota 2 (OpenAI (2018)). Due to its popularity, it is widely tested, and numerous readily used implementations exist online.

As with all Actor-Critic methods, PPO consists of a policy and a critic. Broadly speaking, the policy is tasked with choosing an action depending on the state, and the critic is tasked with assessing how good this action was. The main difference between PPO and other Actor-Critic methods, however, is how the PPO updates its policy. It tries to be conservative in how quickly it changes the policy during training, in order to increase stability of learning. PPO also works great with both continuous and discreet action spaces, as well as multi-action action spaces, as touched upon earlier.

Reasons for why we chose PPO specifically is outlined in section 4.5.4.

Chapter 3

Related works

Since its first appearance in 1962, BO has been extensively researched (Garnett (2023)). In more recent times, interest in cost-aware BO has garnered more interest. Swersky et al. (2013) introduced a cost-aware variant of the entropy AF, which paved the way for modifying EI into EI per unit cost (EIpu):

$$\text{EIpu} = \frac{\text{EI}(x)}{c(x)} \tag{3.1}$$

where EI is Expected Improvement and $c(x)$ is the underlying cost function. EIpu is an AF designed to balance the tradeoff between evaluation cost and expected utility of the evaluation. Snoek et al. (2012) showed that this can increase the performance of BO on a variety of problems. This result was called into question by Lee et al. (2020), which found that EI performed better than EIpu on nine of their twenty test cases. They reason that this is because EIpu takes cost too much into consideration during the later parts of the optimization run, where you want to act greedy, and they argue that EIpu is only better when the optimum is located in a cheap area of the input space.

To mitigate this, Lee et al. (2020) developed a method called Cost Apportioned BO (CArBO), that deals with this issue by reducing the impact of the evaluation cost as the budget depletes. They do this by introducing a new AF, named EI-cool, which is defined as follows:

$$\text{EI-cool}(x) = \frac{\text{EI}(x)}{c(x)^{\alpha}}, \alpha = (\tau - \tau_k)/\tau$$

where EI is Expected Improvement, $c(x)$ is the underlying cost function, τ is the optimization budget, and τ_k is how much of the optimization budget has been currently used. They conclude that EI-cool outperforms EI and EIpu on an extensive set of real-world benchmarks and that it can be easily extended to work with other AFs, not just EI.

There has also been research done on attempting to learn AFs from data, which in this case are source tasks. A source task is an optimization problem similar to the one we are currently trying to solve. Wistuba et al. (2018) proposed a method they name transfer acquisition function (TAF), which improves performance by making the AF a superposition between the GP's prediction on source tasks, and the target task, i.e. our current optimization problem.

Volpp et al. (2020) then expanded upon this idea by learning a AF from a set of objective functions that are similar to the true objective function we want to optimize later. The difference is that instead of explicitly weighting between the source and target tasks, they instead directly learned this weighting using RL. They introduced the method MetaBO, a method this thesis is heavily based on. MetaBO is a method for learning acquisition functions using RL in order to exploit prior knowledge. They show that their method consistently outperforms current methods, and is broadly applicable to a wide range of different problems.

MetaBO does not take cost awareness into account, and so this thesis is concerned with extending the research done by MetaBO into cost-aware BO. To the best of my knowledge, there are currently no other scientific works that combine the use of RL in order to learn an AF in a cost-aware scenario.

Chapter 4

Method

In this chapter, I introduce a method I have named **C**ost-**A**ware **R**einforcement **Learning** based **Bayesian Optimization** (CARLBO), which is aimed at learning an AF that takes advantage of knowing the cost of evaluating the function at a given input x_i in order to increase performance. This distinctive feature allows the AF to manage the optimization budget more effectively, leading to improved data efficiency. By only learning the AF, rather than reformulating the entire BO framework, I attempt to make improvements upon established Bayesian Optimization techniques. In essence, this new method is a combination of previously developed methods introduced in chapter 3. That is, learning an AF like done by MetaBO, and taking the cost-aspect into account like done by Lee et al. (2020).

As the AF we want to learn is simply a mapping from some input information to a single measure of utility, there are multiple ways we can model this. In this thesis, we use a neural network to represent the function. The reasoning will be laid out in subsection 4.5.1. This AF is learned using the PPO algorithm for reasons stated in subsection 4.5.4.

In this chapter, I will first talk about the advantages of using RL in order to learn an AF, and how we can model the BO as an RL problem. Then, I will introduce my method and give an algorithm for how it works, as well as talk about what information my AF utilizes. At last, I will talk about the specifics of some implementation details of the method in how it is trained.

4.1 Advantages of Learning an Acquisition Function Through Reinforcement Learning

Using RL to learn an AF offers various advantages. For one, an RL-based AF can use any additional information that may be underutilized by traditional, handcrafted AFs. While the latter are sculpted based on theoretical guarantees and are often simplistic in their nature, an RL-based AF has the potential to incorporate any relevant data provided, increasing its performance.

Traditionally, handcrafted AFs have used additional information in relatively simple ways, in comparison to more sophisticated methods. As an example, a common method to incorporate cost awareness is EIpu, as detailed in chapter 3, which is a method that scales EI by the inverse cost. The advantage of using RL to learn an AF is that it can directly utilize such additional information when making a decision. Unlike traditional, handcrafted methods, RL-based methods can potentially leverage this information in a more nuanced way, hopefully leading to increased performance.

One of the biggest advantages of using RL is its capacity for class-specific learning. Unlike traditional AFs, which are created to perform well across a broad spectrum of function classes, RL-based AFs can be trained on specific types of functions, extracting useful information about the underlying structure. While this can hurt the generalization of the AF, it can possibly gain a significant boost when used on problems of the same type as it was trained on. This can then be exploited if we have domain knowledge about the underlying objective function by training on similar function types.

4.2 Modeling Bayesian Optimization as a Reinforcement Learning Problem

One might consider designing an RL agent that directly outputs new coordinates based on the points evaluated so far, along with their corresponding function values. However, by framing the problem within the BO framework, we significantly simplify this process. BO is a tried and tested method that has demonstrated effectiveness for the type of functions considered in this thesis. By leveraging this framework, we retain the beneficial properties inherent in BO, with the hope of enhancing the performance of an already well-performing optimization strategy.

Reinforcement Learning	**Bayesian Optimization**
Policy π	Aquisition Function
Episode	Optimization run on $f \in F_0$
Episode length T	Optimization budget T
State s_t	Information available to the learned AF
Action a_t	Query point x_t
Reward r_t	Negative simple regret $-R_t$
Transition $p(s_{t+1}\|s_t, a_t)$	Evaluation of objective function f, update of information

Table 4.1: Concepts in Reinforcement Learning and their counterparts in Bayesian Optimization. Adapted from Table 1 in Volpp et al. (2020).

The modeling of the BO problem as an RL problem is helped by the numerous similarities shared by the two frameworks, the most significant being that they are both sequential decision-making problems. As can be seen in Table 4.1, the various concepts in RL have direct counterparts in the BO framework. We can thus train the AF as an RL agent by conducting optimization runs on randomly sampled functions.

4.3 CARLBO Algorithm

In order to learn an AF, we need a set of objective functions, denoted $\mathcal{F}$, that we can train the RL agent on. These objective functions need to be cheap to evaluate and should ideally share structural similarities with the true objective function we wish to later utilize this learned AF on. Doing this allows the learned AF to take advantage of the inductive bias introduced through the choice of $\mathcal{F}$. However, as shown by MetaBO, the performance on unseen function types should fall back to that of more common AFs even if used on function types not present in $\mathcal{F}$.

As shown in table 4.1, an episode in the RL setting is just an optimization run on a function. Every timestep t, the agent receives state s_t, which is information about a set of points in the input space as detailed in section 4.4. The agent then chooses an action a_t, which is then treated as the next query point by the RL environment. The cost of evaluating the function at a_t is then subtracted from the remaining budget, and the GP is updated with the new data. The environment then returns s_{t+1} and r_t, and the whole process is repeated until the evaluation budget is depleted.

CARLBO, an algorithm for learning a cost-aware AF using RL is presented in Algorithm 2. Note that after training, the learned AF is fully described by the neural network, which means that we can discard the entire RL framework. Only the weights and architecture of the trained network is needed in order to use the learned AF. By using differentiable activation functions in the neural network, we allow for the of use gradient-based methods to maximize AF, which is an important step in the BO loop.

Algorithm 2 CARLBO

```
1: Choose a set of objective functions $\mathcal{F}$
2: Initialize the agent and environment $\mathcal{E}$
3: Train the agent using PPO by doing optimization runs on $f \in \mathcal{F}$.
4: Discard entire RL framework except for learned policy/AF
5: Run an optimization procedure on the true objective function $f^*$ using BO with the
   learned AF
```

4.4 Input Variables For the Learned Aquisition Function

Giving the learned AF enough information to make informed decisions is important to maximize its performance. In this section, I will go over the different input variables passed into the AF and what their importance is. The first three inputs are local information, meaning that they say something about a specific point x_i. The last three concern global information, meaning that they say something about the state of the entire system, and thus are the same for any point x_i.

- **Mean value of the Gaussian Process for point** x_i The mean value of the GP for point x_i is expected value of that point according to the GP. It informs the AF about the predicted value for this point, which in turn plays a large role in determining how good this point is.
- **Standard deviation of the Gaussian Process for point** x_i The standard deviation of the GP for point x_i is a measure of uncertainty for this point according to the GP. In general, the further away from a previously sampled point x_i is the higher the standard deviation. The standard deviation is useful for guiding exploration (points with very low standard deviation are often less interesting to explore, as we are quite certain about their value).
- **Evaluation cost for point** x_i The evaluation cost for points x_i is important information to consider when deciding if x_i is useful to sample. The cheaper the evaluation cost, the more interesting sampling this point should be, as it will allow

for more samplings over the course of an optimization run. Adding this information to the learned AF is the main contribution in this thesis.

- **Remaining evaluation budget** The remaining evaluation budget denotes how far in the search process we currently are. This information can be useful to guide the exploration vs. exploration trade-off, as it is natural to think that this should be dependent on how many potential evaluations we have remaining.
- **Best found solution so far** The best-found solution so far is another global variable. It denotes the goodness of the best solution found so far, which is very useful when compared to the mean value and standard deviation of the GP. If the mean value for x_i is above the best found solution so far, it is potentially a very interesting point to sample. This information allows potentially allows us to generalize to functions of different scales, as we can always compare the mean and standard deviation at a point x_i to the best found solution so far.
- **Most expensive evaluation cost** This global number denotes how much the most expensive evaluation costs. This is included in order to give the AF a baseline to compare the cost in each single point against.

4.4.1 Dimension dependent information

Unlike MetaBO, I do not include information about the coordinates of the point x_i. Whereas their main focus is on using transfer learning to learn an AF to use on a specific objective function, my aim in this thesis is to make a general AF that works on a variety of problems. The addition of the coordinates for x_i means that the AF is no longer dimension-agonistic. That is, the size of the input is dependent on the dimension of the objective function. That means that an AF trained on two-dimensional functions will not work for any other dimensions. By omitting this information, our learned AF can be trained on one size and used on others. While not in their main method, MetaBO showed that omitting this information still allows for a good AF to be learned.

4.5 Details of training

Due to the nature of this problem, it is not entirely straightforward to apply common RL algorithms out of the box. In this section, I will go into some of the details of the training process. I will first talk about how specifically we model the action space, as this is where the problem lies. I will then justify my choice of using PPO use to train the agent.

4.5.1 Representing the Learned Aquisition Function

How we choose to represent the learned AF is of great importance, both when it comes to the training process as well as using it after training. The two main approaches introduced in subsection 2.3.4, tabular methods and function approximation, each has their benefits. Tabular methods provide theoretical convergence guarantees, which are advantageous. However, to apply them in our context of continuous state space, that is the values denoted in section 4.4, we must discretize them. This discretization, particularly when dealing with large input spaces, results in information loss due to the necessary coarseness, compromising the detail and precision of the learned AF.

A function approximation approach avoids the need for discretization. This provides a more flexible and potentially more accurate representation of the learned AF. For neural networks specifically, given their known capacity to approximate complex continuous functions, we can approximate the desired AF with high accuracy, given a sufficiently large network. While they lose guarantees of converging to the optimal policy in the context of RL, they have proven performance in RL settings across many different types of problems.

Additionally, using a neural network allows for the use of gradient-based optimization methods to maximize the AF when it is in use. The prevalence of neural networks in machine learning has resulted in robust, user-friendly tools and libraries, making implementation fast and easy. Therefore, despite tabular methods being a potentially viable option, function approximation using neural networks was selected for this thesis due to its expected ability to provide a more precise and flexible representation of the learned AF.

4.5.2 State

In terms of BO, our system is fully described by the GP, the cost function, and the global information, as mentioned in section 4.4. When it comes to RL, however, we need to evaluate a discreet amount of points, in order to make it manageable for the PPO algorithm.

We do this by choosing a set $\mathcal{X}$ of points in the input space, and getting the values from the GP and cost function in these points. Formally, the state in time t is then

$$s_t = [\mu_t(\mathcal{X}), \sigma_t(\mathcal{X}), T_t(\mathcal{X}), b_t, f(x_t^+)]$$

where $\mu(x)$ is the mean function of the GP, $\sigma(x)$ is the standard deviation function of the GP, $T(x)$ is the cost function, b_t is the remaining evaluation budget, and $f(x^t)$ is the best found solution so far. Note that if we for example choose $\mathcal{X}$ to be of size 100, that is we have 100 points in the input space, the state will be of size $[100, 6]$. We thus have information about 100 different points in the input space.

We are free to choose how we create this set of input points $\mathcal{X}$. The freedom to choose how the input space is discretized means that it can be adapted to the problem at hand. It can be a uniform grid, it can be randomly sampled coordinates from the input space of the objective function, or any other suitable approach.

An optimal strategy for choosing the set $\mathcal{X}$ might follow a procedure similar to the MetaBO method, where a Sobol sequence is used to generate a grid of points. These points are then evaluated using the AF we are currently learning, and a second, more refined grid is constructed close to where the AF predicts maximum utility. This allows the creation of a set $\mathcal{X}$ that is focused on more promising regions of the input space, while still keeping it sparse enough for efficient computation. However, this technique was not adopted in this thesis due to the complexity of implementation. Specifics of how $\mathcal{X}$ was chosen in this thesis will be specified in chapter 5.

4.5.3 Modeling the Policy

In a conventional RL setting, the action space is usually formulated as either a continuous or discrete space, and the agent's decision is directly related to the entire state. However, in CARLBO, I adopt a different approach that more closely mirrors the behavior of a conventional AF.

Typically, we feed the entire state into our function approximator and receive the action, or coordinate in our case, as an output. In order to mirror the behavior of a conventional AF, we instead input each individual point x_i in the state into our model and get one value as a prediction. Then, if the set of input points $\mathcal{X}$ is of size 100, we predict 100 distinct values, one for each point. By turning the collection of these points into a categorical distribution, we make sure that the value predicted by the AF for any point x_i represents the desirability to sample at this point. By sampling a point from this categorical distribution, we end up with our action.

The formal definition of the policy is then as follows:

$$\pi_\theta(\cdot|s_t) \equiv \mathrm{Cat}\left[\alpha_\theta(x_1), \dots, \alpha_\theta(x_N)\right] \tag{4.1}$$

where π denotes the policy, s_t is the state at time t, Cat is the categorical distribution, α_θ is the parameterized AF we are learning, and x_i denotes a point in the input space.

4.5.4 Selection of the Reinforcement Learning Algorithm

Creating a good AF using RL is possible with a myriad of different RL algorithms, both model-based and policy-based. Despite the numerous options available, I have opted to use the PPO algorithm in this thesis.

The primary goal in our context is to correlate the output of the learned AF with the value of sampling a particular point in the input space. Both value-based methods and policy-based methods can do this. Policy-based through predicting a probability for each point, and value-based methods through predicting the value of being in the state. Both these should very well correlate with our primary goal. One of the inherent problems with value methods is they don't inherently have exploration built in. However, with the way we model our action space, this issue is circumvented as we still sample from the categorical distribution.

Despite both methods being suitable for this task, I have selected PPO for a concrete reason. It has already been demonstrated by MetaBO to be highly effective for this specific task. Therefore, I chose to build on this proven success, making it the primary motivation behind using PPO in this study.

Reward

For the reward function, I have chosen to use the negative logarithm of simple regret as the reward. This is calculated as the difference between the true maximum value of the function and the best value that the agent has found so far. By taking the logarithm, we ensure that the closer we are to a maximum, the more significant a small improvement is[1]. Furthermore, using the negative sign aligns with the agent's goal to minimize regret.

[1]The improvement from 80% of the maximum value of the function to 80.5% is not very significant, but an improvement from 99% to 99.5% is very significant

The reward function is crucial in RL as it shapes the agent's learning process, and defines its goal. In our case, we can either assign rewards at the end of each episode or after every timestep. However, considering the variable length of episodes in our setup, providing rewards after every timestep may encourage the agent to extend the episode length unnecessarily, instead of focusing on finding the maxima.

Thus, to drive the agent toward efficient exploration and exploitation, we choose to only reward the agent at the end of each episode. This approach makes sure that the only goal of the agent is to have come as close as possible to the maximum of the function within the optimization budget, which is perfectly aligned with our goal.

Chapter 5

Experiments

To assess the effectiveness of CARLBO, I have set up a set of experiments designed to test its capabilities in different scenarios. In these experiments, I trained two distinct agents, and I test these on various objective functions in order to understand the method's performance under different scenarios. Alongside these agents, I have also tested several benchmark methods to ensure a robust comparison. This chapter will outline the experimental process, starting with the standard settings and configurations shared across all experiments, and subsequently moving to introduce each individual experiment.

5.1 Experimental Setup

These experiments were executed on Python 3.10.8, using an RTX 4090 GPU. As the environment I have developed heavily utilizes the GPU in order to run many episodes in parallel, a high-performance GPU with a large amount of video memory is crucial in order to achieve fast training times. In the rest of this section, I will introduce the elements that are consistent across all experiments.

5.1.1 Environment

In this subsection, I will first outline one of my contributions: developing an environment[1] for accelerating learning AFs using parallel computations on a GPU. I will then go through

[1]The environment is available at https://github.com/ALjone/Master-Thesis

some experiment-specific choices made in the environment, namely choices related to the surrogate model, the action space, and the reward function.

A key contribution of this thesis is the development of an advanced RL environment using Python for training an AF. This environment is highly extensible, allowing the integration of any objective function through the extension of a base class.

The environment utilizes GPyTorch, a GPU-accelerated library for Gaussian Processes (GP) training, as introduced by Gardner et al. (2018). Due to GPyTorch's batch-training capabilities, we can run thousands of objective functions in parallel. A crucial feature of this environment is that all operations are GPU-accelerated, and all Pytoch tensors are hosted on the GPU, removing the need for moving tensors back and forth between CPU and GPU during training, leading to significant speed improvements. In essence, this GPU-based environment makes training RL based AFs much faster compared to traditional CPU-based environments.

As seen in table 5.1, doing 2048 optimization runs in parallel on the GPU significantly outperforms all other variants. We observe an approximate 5x speedup compared to the parallel CPU version. This allows for much faster training, allowing for either longer training, or more experimentation. One factor not shown here is that when training the agent, we need to have the tensors on the GPU. Having the tensors already on the GPU removes the overhead of moving them, which can be costly.

Number of Objective Functions	Environment	Steps per Second
2048	GPU	1700
2048	CPU	330
1	GPU	45
1	CPU	65

Table 5.1: Comparison of speed performance of the environment. Note that CPU beats GPU when only running one objective function. This is likely due to overhead in moving tensors from CPU to GPU.

Initial Design

When fitting a GP, there need to be some initial points that have already been sampled. Lee et al. (2020) proposed a method to do a cost-effective initial design. While this can be very useful when using the AF, I chose to instead randomly sample points from the input space. This was done due to the simplicity of implementation, as well as the fact that the cost-effective initial design can be too computationally costly for a training loop where speed is crucial.

Another important choice was that I don't subtract the cost of the inial points from the initial budget. This is done because the initial design is out of the control of the agent, and subtracting the cost of sampling the initial design is in practice the same as starting with a randomized budget, which I already do.

In all experiments, 3 points are randomly sampled from the input space before the optimization run is started.

State Grid

As mentioned in subsection 4.5.2, I discretize the input domain into a set $\mathcal{X}$ of points. In our case, this is constructed as a uniform grid. This implies that at each timestep, the agent can select any point $x_i \in x$ to sample next. This is a straightforward and easy-to-implement method of choosing $\mathcal{X}$, which is also computationally fairly cheap.

Theoretically, the agent could benefit from a more refined step $\mathcal{X}$. One that is denser close to the maxima of the function, such as the one proposed by MetaBO and detailed in subsection 4.5.2, is a good candidate. However, the implementation of such a refined action space might introduce additional computational complexity or require more advanced techniques, which is outside the scope of this thesis. Therefore, despite its downside, I opted for a uniform grid-based set $\mathcal{X}$ for its ease of implementation.

One of the main disadvantages of this is that we need a high resolution, i.e. the number of points in each dimension, in order to have enough density close to maxima. If we have a resolution of for example 60, this is equal to $60^2 = 3600$ in 2D, and $60^3 = 216000$ in 3D. One must therefore need to be careful when choosing the resolution.

5.1.2 Training

The training loop in these experiments was conducted using a modified version of cleanRL, a framework developed by Huang et al. (2022), which has single-file implementations of many different reinforcement learning algorithms.[2]

[2]The specific file my training loop was based on can be found at cleanRL's Github repo [link]

Note that opposite to MetaBO, I fit a GP to the data in every timestep, whereas MetaBO used a fixed length scale and variance for their kernels, circumventing the need for learning these parameters from the data.

The hyperparameters used for PPO are consistent across all experiments. They are displayed in table 5.2.

Hyperparameter	**Value**
Total Timesteps	5,000,000
Anneal Learning Rate	True
GAE Lambda	0.98
Number of Minibatches	4
Update Epochs	4
Normalize Advantage	True
Clip Coefficient	0.2
Value Function Coefficient	1
Max Gradient Norm	0.5
Target KL Divergence	0.3
Entropy Coefficient	0.01
Number of Steps	32
Learning Rate	1.0×10^{-4}
Discount Factor (γ)	1

Table 5.2: Hyperparameters used for the Proximal Policy Optimization (PPO) algorithm.

These hyperparameters are consistent with the ones used by MetaBO, with some small differences. As I only reward the agent at the end of the episode, I set the discount factor γ to 1, as all steps of the optimization process are equally important. I also have a lower number of steps (i.e. rollout length) compared to conventional PPO implementations such as the one suggested by Huang et al. (2022), which is due to our environment having short episodic lengths, with the length generally being around 20 steps.

5.1.3 Policy and Network Architecture

The function approximation I use to represent our AF is a simple feed-forward neural network with 7 hidden layers, each consisting of 32 neurons. The input layer has 6 neurons, while the output layer consists of a single neuron. This structure results in a network composed of 15 107 parameters, significantly smaller than the network used by MetaBO (approximately 161 000 parameters. My choice of using a smaller network was motivated by the desire to reduce computational costs. During testing, smaller networks appeared to perform as well as larger networks, thus leading me to choose a smaller network without any notable drawbacks. The activation function of choice was Leaky ReLU, and I use the softmax function in order to transform the outputs of the network into probabilities.

In addition to this, the agent has one parameter that controls the temperature of the softmax function when creating the categorical distribution. This is done in order to aid exploration early on. While the network of the agent in a normal RL scenario predicts all the logits individually, our agent predicts one logit at a time, making it harder to increase exploration without simply lowering the size of the output of the network across the line. [3]. In order to aid the agent with doing this, I give it a temperature parameter it can control. The logits predicted by the network, i.e. the utility of sampling at each point, is divided by this temperature before being sent to the softmax function. The temperature starts at a value of 3, in order to facilitate early exploration.

5.1.4 Evaluation Cost in Experiments

In our experiments, each objective function is associated with an evaluation cost function, which assigns a cost to sampling a point x_i in the input space. Each objective function is also associated with a total evaluation budget, which limits the number of times the objective function can be sampled from. For each optimization run in the training loop, I sample both an underlying evaluation cost function as well as an evaluation budget. The evaluation budget is sampled from a uniform distribution $T \sim U(a, b)$, where a and b are the lower and upper bounds of the evaluation budget.

The cost functions chosen in our experiments are polynomial in nature, characterized by a polynomial term for each dimension in the function space. This results in a non-linear function that can be formally defined as:

$$c(\mathbf{x}) = \sum_{i=1}^{n} m_i (x_i + 1)^{p_i} + b$$

Here, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ represents the input vector, $\mathbf{m} = (m_1, m_2, \ldots, m_n)$ are the coefficients for each dimension, $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ are the exponents for each dimension, and b is a constant term. This function is evaluated on $x_i \in [-1, 1]$ for the i^{th} dimension.

[3]This is due to how the Softmax function works. The logits [3, 3, 4] would result in a distribution with the mass fairly evenly distributed, whereas the logits [30, 30, 40] will put nearly all the mass in the logit with value 40, despite the ratio between the logits being the same

To generate a variety of polynomial cost functions, a sample is generated for each coefficient m_i, each exponent p_i, and the constant term b from uniform distributions. The specific distributions depend on the experiment and are given in their respective sections.

I chose the polynomial cost function outlined above, as it represents a lot of real-world scenarios. Polynomial scaling depending on some variables is not uncommon for a lot of algorithms when doing hyperparameter optimization, such as the number of parameters in a layer scaling quadratically with the hidden layer size.[4] This also allows us to have close to linear functions when the exponents sampled are close to 1. By sampling the coefficients, exponents, and constant terms, I train the agent on a diverse set of evaluation cost functions, hopefully improving robustness.

Across all experiments, unless otherwise specified, the values used to sample every cost-related parameter were shown in table 5.3

Parameter	**Lower Bound**	**Upper Bound**
Evaluation Budget	500	800
Constant Range (b)	5	10
Linear Range (m_i)	10	20
Polynomial Range (p_i)	0.5	1.5

Table 5.3: Evaluation cost parameter ranges for all experiments unless otherwise specified.

These parameters were chosen to give a wide range of possible evaluation cost functions in order to create a robust and general AF. With these parameters, if sampling at random, the average optimization run samples approximately 18 times.

5.1.5 Benchmarks

In my experiments, I utilize several benchmark methods as benchmarks for comparison:

- **Random:** The simplest benchmark involves randomly sampling a point in the input space at each timestep. While naive, this method is an essential benchmark that provides a lower bound on performance.

[4]Assuming that the input and output of the layer has the same size

- **Expected Improvement (EI):** Given its wide usage and straightforward implementation, EI, a commonly employed AF, serves as a valuable standard benchmark. Its consistent performance and prevalent use in the field make it an important reference.
- **EIpu** EIpu takes evaluation cost into account in a very naive way, by scaling EI with the evaluation cost in point x_i. This is a commonly used method in these cases and provides a naive benchmark that still considers evaluation cost.
- **MetaBO** I train an agent from scratch using the method proposed in Volpp et al. (2020) [5], on the exact same setup as my AF is trained. This is essentially using CARLBO, but omitting any information about the cost of sampling.
- **EI-cool** EI-cool builds upon EIpu, and introduced a cooling factor on the scaling. The longer the optimization process goes, the closer it resembles EI. This method is designed to be a more robust version of EIpu, and was shown to achieve good results by Lee et al. (2020)

These benchmarks should provide a robust collection of benchmarks I can compare CARLBO to, in order to get a fair assessment of its performance.

5.2 Experiments

In conducting these experiments, I train two different AFs according to algorithm 2. The first AF I train will be denoted $\alpha_{\text{CARLBO MM}}$ and is trained by choosing $\mathcal{F}$ to be a set of 2-dimensional multimodal functions with randomly drawn parameters. The second AF, denoted $\alpha_{\text{CARLBO GP}}$[6], is trained by choosing $\mathcal{F}$ to be a set of modified 2-dimensional Goldstein-Price functions. These functions will be introduced in the following subsection. In addition, I trained $\alpha_{\text{MetaBO MM}}$ and $\alpha_{\text{MetaBO GP}}$, which is trained the exact same way, but without evaluation cost information. Note that both AFs were trained with the resolution of the set $\mathcal{X}$ of input points set to 60 in each dimension.

I test the agents both when sampling from the policy, i.e. by turning the outputs into a categorical distribution and sampling from this distribution, as well as when greedily taking the best action. This is done due to the fact that the policy learns to condition its

[5] I omit any positional info as input, as done in some of their experiments, for the same reasons as outlined in section 4.4.

[6] GP here is Goldstein-Price, not to be confused by GP

output on the state assuming that we follow the policy for the rest of the episode. Since the policy was trained with sampling, this is what it is conditioned on.

These two models are then tested in different settings in order to assess the performance of the model both on seen and unseen function types, in order to test both its capability of learning specific function types, as well as its generalization capabilities. As the idea behind $\alpha_{\text{CARLBO GP}}$ is to test CARLBO's function-specific learning capabilities on functions with a very specific structure, most of the experiments will only be run using $\alpha_{\text{CARLBO MM}}$, as the performance of $\alpha_{\text{CARLBO GP}}$ is expected to be poor.

In this section, I will first introduce the different objective functions that are used in the experiments. I will then introduce all the experiments that were done.

Note that in all experiments, all evaluations of the functions have no noise. I also frame all experiments as maximization problems, although the method should work equally well for minimization problems. All functions, when drawn, are scaled to have their maxima be a random number drawn from $U(\frac{1}{3}, 2)$ in order to help generalization.

5.2.1 Objective Functions

All experiments conducted in this thesis are conducted on variants of multimodal functions, noisy convex functions, or the Goldenstein-Price function, all of which will be introduced in the sections following. The multimodal function type should in theory be easy to optimize and has multiple easily identified local maxima. The noisy convex functions are strongly convex but with added Gaussian noise. Goldstein-Price represents a type of function with very strong properties, that are generally harder to optimize. It is also a commonly used benchmark for testing optimization methods and was used by MetaBO.

Multimodal Functions

In order to test that this method works, I construct a scenario where one would assume that the inclusion of evaluation cost is highly relevant. As seen in figure 5.1, this is a function with 4 peaks, each in its own quadrant.

These multimodal functions are constructed as Gaussian Mixture Models, and in this case, there are two Gaussians in each dimension. They are given by the function:

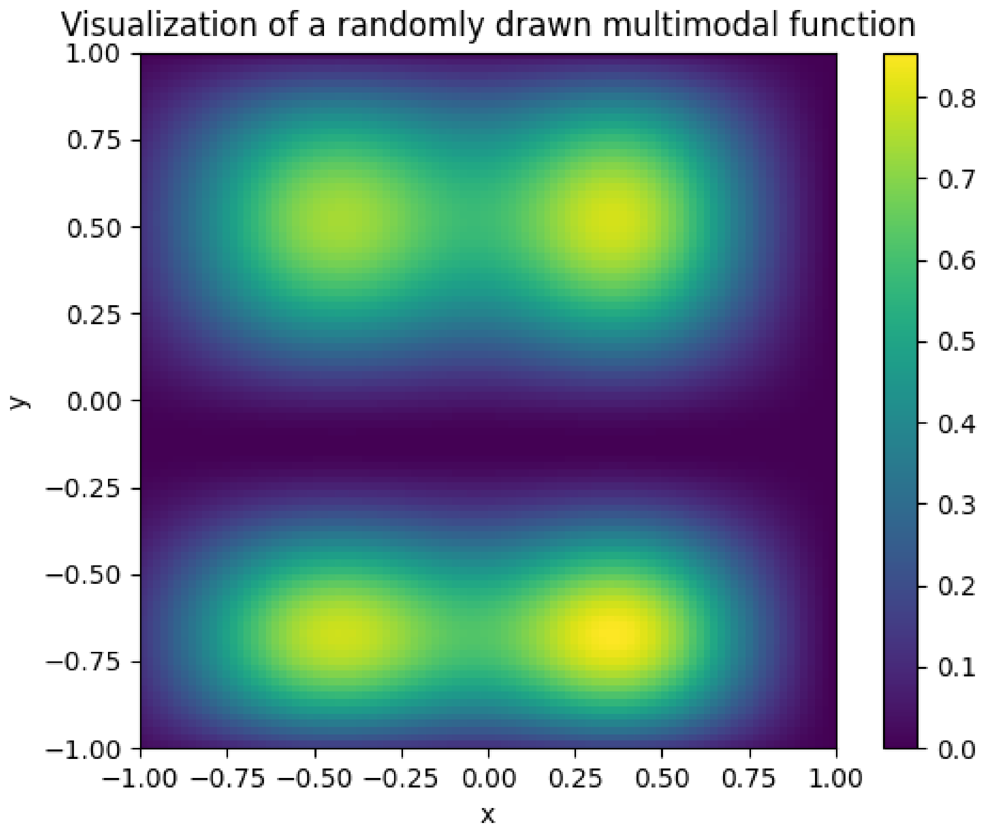

Figure 5.1: A visualization of a multimodal function with 4 local maxima and one global.

$$f(\mathbf{x}) = \sum_{i=1}^{n} [A_{1i} \cdot e^{-\lambda_{1i} \cdot (x_i - \mu_{1i})^2} + A_{2i} \cdot e^{-\lambda_{2i} \cdot (x_i - \mu_{2i})^2}]$$

where:

- **n** is the number of dimensions in the function
- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is the input vector.
- A_{1i}, A_{2i} are the amplitudes of the Gaussians in the i^{th} dimension.
- λ_{1i}, λ_{2i} are the decay rates of the Gaussians in the i^{th} dimension.
- μ_{1i}, μ_{2i} are the means of the Gaussians in the i^{th} dimension.

Here, for each dimension i, two Gaussian functions are defined with different parameters (amplitude, decay rate, and mean), and the sum of these Gaussians is the value in that dimension. The overall value of the entire function in a point x_i is the sum of the values in all dimensions. The specific ranges for how these values are sampled are given in table 5.4. Note that as the amplitudes are also randomly sampled, there is potentially only one global maximum, but on average the local maxima are close to the global maximum in terms of peak. These functions are evaluated evaluated on $x_i \in [-1, 1]$ in the i^{th} dimension.

Parameter	Range
Amplitude (A)	[0.8, 1.2]
Mean (μ_1)	[-0.85, -0.15]
Mean (μ_2)	[0.15, 0.85]
Standard Deviation (σ)	[0.15, 0.5]

Table 5.4: Parameter ranges for the multimodal functions

These functions, in the example of the 2D case, have one peak in each quadrant. By using the evaluation cost information, the learned AF will hopefully learn to first explore the cheaper of the quadrants, before moving on to the more expensive ones. Thus I construct a scenario where our model should work well, which is done in order to highlight the strengths of the method.

When optimizing the multimodal functions, I use the spectral mixture kernel. This is due to the fact that the amount of mixtures in each dimension is known (two in each dimension), and the spectral mixture kernel is better equipped to model these functions than the RBF kernel, especially when dealing with small numbers of data.

Noisy Convex Functions

Convex functions represent a distinct class of optimization problems. They are generally easy to optimize due to the fact that local maxima are also global maxima. To make this more challenging, I add correlated Gaussian noise to the functions, which in turn makes them slightly harder to optimize. This noise can be seen in figure 5.2[7]. The function is thus similar to the multimodal one in some ways. They are both easy to optimize, and the multimodal functions are, in the 2D case, convex in each quadrant. However, they have different challenges when it comes to optimization. Multimodal functions have the challenge of finding the correct mode, whereas the noisy convex functions have considerably more local maxima, and the surface must be navigated in a different way.

[7]Note that the evaluations themselves are still deterministic, the function is just a sum of a convex function and some precomputed Gaussian noise

The inclusion of noisy convex functions is thus motivated by these slight differences. It provides a good way to test if the performance of the learned models translates to a similar, yet different type of function.

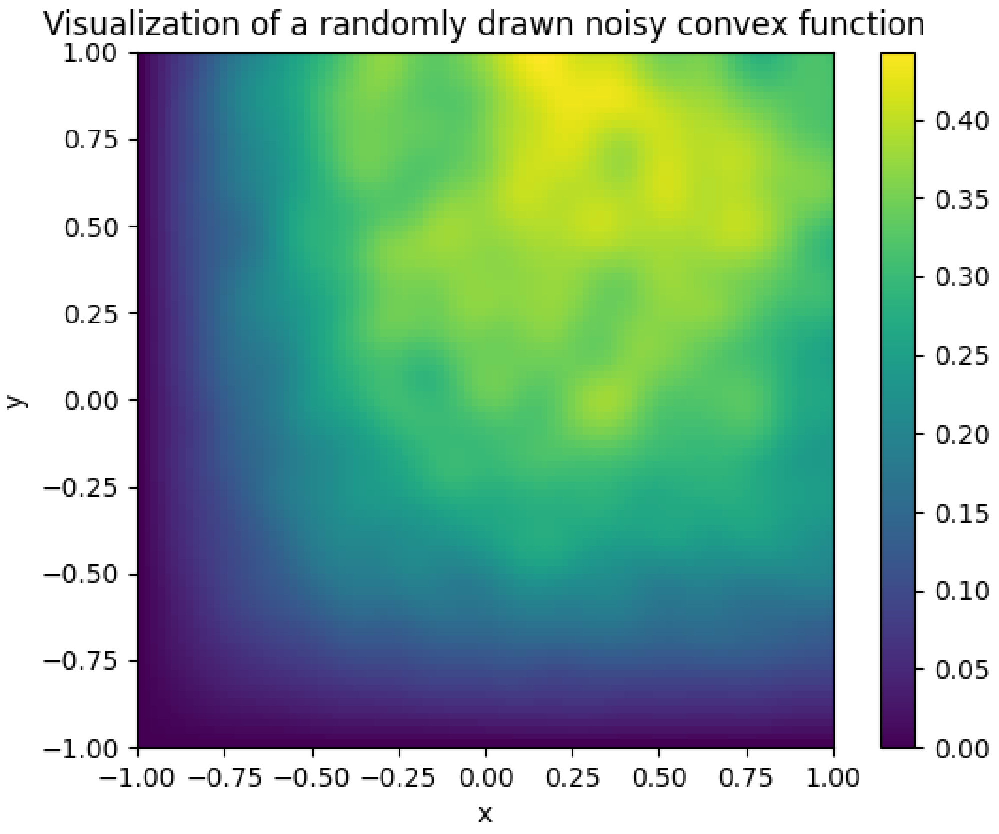

Figure 5.2: A visualization of a noisy convex function.

$$F(\mathbf{x}) = \left(\prod_{i=1}^{n} a_i (x_i - p_i)^2 e^{-b_i (x_i - p_i)} \right) \cdot N$$

where:

- **n** is the number of dimensions
- $\mathbf{x} = (x_1, x_2, ..., x_n)$ is the input vector.
- a_i, b_i, and p_i are random parameters associated with the i^{th} dimension.
- N is a noise term generated by a Gaussian function.

The term N is generated by a Gaussian Filter with $\mu = 1$, $\sigma = 1.5$, and correlation $=$ 3.5.[8] These values were chosen through visual inspection of the functions. These functions

[8]This was implemented using the gaussian_filter method in SciPy.

are evaluated evaluated on $x_i \in [-1, 1]$ in the i^{th} dimension. The specific ranges for the sampled values are given in table 5.5

Parameter	Range
a_i	[-0.5, -0.01]
b_i	[0.01, 0.5]
p_i	[-1.1, 1.1]

Table 5.5: Parameter ranges for the noisy convex functions

When optimizing the noisy convex functions, I use the RBF kernel. The RBF kernel is well suited to a convex problem like this and is commonly the go-to kernel.

Goldstein-Price

The Goldstein-Price function (Surjanovic and Bingham) is a well-established optimization benchmark characterized by a pronounced structure, as depicted in Figure 5.3. It is significantly different from the multimodal functions and thereby tests the model's generalization capabilities. For the purposes of this thesis, I use the standard form of the Goldstein-Price function but invert it in order to turn it into a maximization problem. I also introduce a random rotation between 0 and 2π radians to generate distinct functions that share the same underlying structure. This is done in order to prevent the AF from overfitting.

The Goldstein-Price function is a two-dimensional function, given by

$$\begin{aligned} f(x_1, x_2) =& (1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)) \\ & \cdot (30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)), \end{aligned}$$

and is usually evaluated on the square $x_1 \in [-2, 2]$, $x_2 \in [-2, 2]$. As shown in Figure 5.3, in this domain, it has two very distinct lines, with a global maximum forming where they cross at $(0, -1)$. This global maximum is also the only maximum of the function.

When optimizing Goldstein-Price functions, I use the RBF kernel. This kernel was chosen as it is the same kernel that MetaBO used in their setup when optimizing their Goldstein-Price functions.

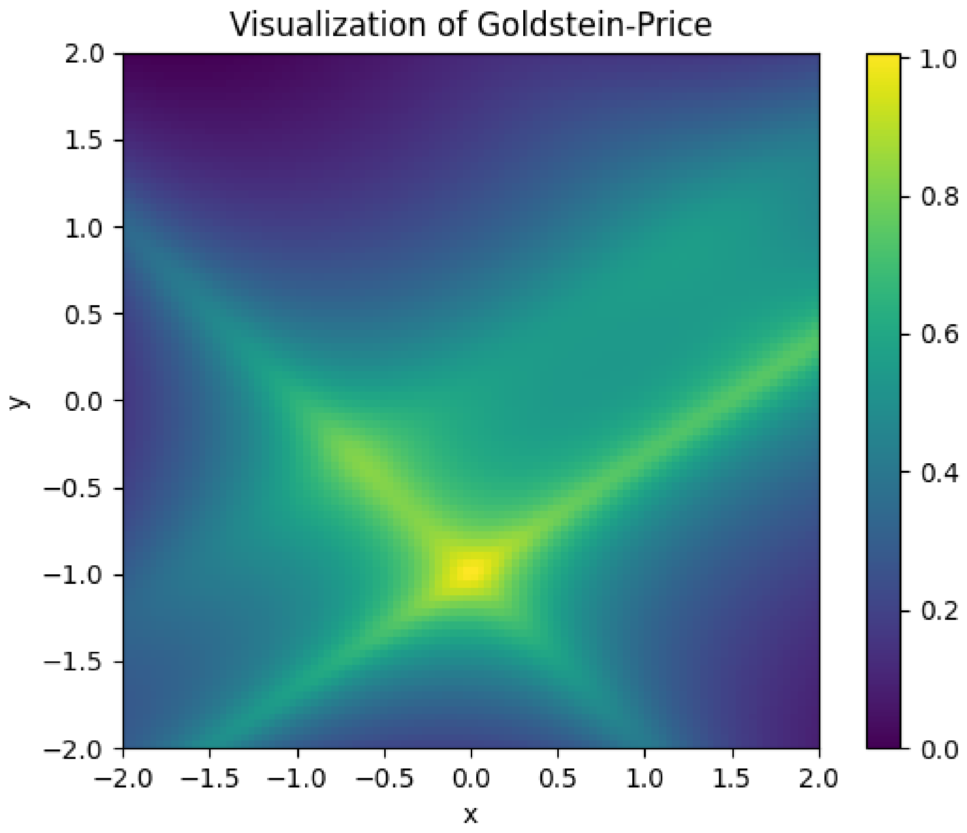

Figure 5.3: A visualization of the inverted Goldstein-Price function with no rotation.

5.2.2 Function Specific Learning

One of the advantages of using RL in order to learn an AF is the capacity to learn patterns in specific functions. In order to test this capability, I designed an experiment where the performance of $\alpha_{\text{CARLBO MM}}$ and $\alpha_{\text{CARLBO GP}}$ are tested on the specific functions they were trained on. Doing this allows me to assess the function-specific learning capabilities of CARLBO, which is an important quality when using the AF for transfer learning.

In these experiments, the resolution was 60 in each dimension, totaling $60^2 = 3600$ points. This offers a fine enough grid of points to reasonably optimize these functions, i.e. there is a fine enough grid of points close to the maxima of the function to reasonably find a good solution.

$\alpha_{\textbf{CARLBO MM}}$ Tested on 2-dimensional Multimodal Functions

In this experiment, I tested the performance of $\alpha_{\text{CARLBO MM}}$ on the exact type of function it was trained on, which is 2-dimensional multimodal functions. This was done in order to test the capabilities of CARLBO to learn an easy-to-optimize function.

$\alpha_{\text{CARLBO GP}}$ Tested on Goldstein-Price Functions

Similarly to the previous experiment, I tested the performance of $\alpha_{\text{CARLBO GP}}$ on the rotated Goldstein-Price functions. I did this to test how well CARLBO performs when tasked with optimizing harder-to-optimize functions with a very clear structure.

5.2.3 Generalization Across Different Dimensions

An important feature for any generalizing AF is its ability to do well on problems with different dimensionality. Due to their inherent design, most AFs are agnostic to the dimensionality of the function being optimized. In order to see if CARLBO has this ability, despite only being trained on 2-dimensional functions, I tested the performance of $\alpha_{\text{CARLBO MM}}$ functions of higher dimensions, namely 3-Dimensional multimodal functions.

In these experiments, the resolution was 20 in each dimension, totaling $20^3 = 8000$ points for the AF to assess. While this is a fairly coarse grid, this is the highest I can go for computational reasons, and should still provide a reasonably sized grid for optimizing 3-dimensional functions.

I conducted two experiments on higher dimensional functions. In the first of the two, the 3rd dimension has an additional evaluation cost associated with it. That is, the cost of evaluating the function for a point x_i is now the sum of the cost in all 3 dimensions. This naturally leads to higher costs in general and higher costs than $\alpha_{\text{CARLBO MM}}$ observed during training. In the second experiment on higher dimensional functions, we omit the cost of sampling in the newly added 3rd dimension. This ensures that the cost of any one point x_i in the input space is consistent with what was observed during training.

$\alpha_{\text{CARLBO MM}}$ Tested on 3-dimensional Multimodal Functions With Two Cost Dimensions

In the first experiment for the higher dimensional functions, the additional function doesn't have any extra cost associated with it. That is, the cost of the evaluation is only dependent on the values of x_i in the first and second dimensions. The idea is that this should more closely resemble the training conditions of $\alpha_{\text{CARLBO MM}}$.

$\alpha_{\textbf{CARLBO MM}}$ Tested on 3-dimensional Multimodal Functions With Three Cost Dimensions

In the second experiment, I tested the performance of $\alpha_{\text{CARLBO MM}}$ on 3-dimensional functions where evaluation cost is consistent across all three dimensions. This is designed to test the capabilities of the learned AF to higher dimensional functions with different underlying cost functions.

5.2.4 Generalization to Different Function Types

Another important quality of a generalizing AF is the ability to function well on a wide variety of function types. In order to test if this is the case with the AFs learned using CARLBO, I tested the $\alpha_{\text{CARLBO MM}}$ and $\alpha_{\text{CARLBO GP}}$ on cost functions different from the ones they were trained on. As these AFs were only trained on one type of functions, the hope is that they excel on the type they were trained on, and worst case fall back to the performance of methods such as EI, as shown by MetaBO.

$\alpha_{\textbf{CARLBO MM}}$ Tested on 2-dimensional Noisy Convex Functions

As I mentioned earlier, the noisy convex functions exhibit some similar properties to the multimodal functions. We therefore test the performance of $\alpha_{\text{CARLBO MM}}$ on these types of functions. Note that due to the ease of optimizing these functions, the resolution of the set of $\mathcal{X}$ input points was set to 100 in each dimension. This is due to the ease of optimizing the noisy convex functions, which calls for a finer grid close to the maxima of the function.

$\alpha_{\textbf{CARLBO MM}}$ Tested on Goldstein-Price Functions

The Goldstein-Price functions exhibit very strong properties and are significantly different from the multimodal functions. I therefore test $\alpha_{\text{CARLBO MM}}$ on these functions, to test performance across different function types.

$\alpha_{\text{CARLBO GP}}$ Tested on 2-dimensional Multimodal Functions

As the Goldstein-Price functions used in this thesis only vary based on one parameter, the rotation angle, we expect that an AF trained on only this function overfits significantly. Nonetheless, we test $\alpha_{\text{CARLBO GP}}$ on 2-dimensional multimodal functions to see if it still manages to generalize to the level of a simple benchmark such as EI nonetheless.

5.2.5 Generalization to Different Cost Functions

Having an AF that performs well across a wide variety of underlying evaluation cost functions is desirable. To test if this is the case with our learned AF, I tested $\alpha_{\text{CARLBO MM}}$ on 2-dimensional multimodal functions with a different underlying evaluation cost function.

This was achieved by transforming the ranges of the sampled parameters for the polynomial-cost function I use in other experiments. The changes made can be seen in table 5.6. The linear part is now significantly smaller, the constant range is higher, and the polynomial range is wider. These values were chosen simply to be significantly different from the ones the AFs were trained on. Note that the evaluation budget remains unchanged.

Parameter	**Original Range**	**Updated Range**
Constant Range (b)	[5, 10]	[15, 20]
Linear Range (m_i)	[10, 20]	[0, 10]
Polynomial Range (p_i)	[0.5, 1.5]	[0.2, 1.8]

Table 5.6: Comparison of original and updated evaluation cost parameter ranges for the generalization across cost functions experiment

Chapter 6

Results

In this chapter, I will go through the specific experiments in the order outlined in section 5.2. I will present the results and comment on what we observe.

In each experiment, length is defined as the number of times the objective function was sampled after the initial design. Part of max is defined as how close the method got to the maxima of the function, and is defined as follows:

$$\textbf{Part of max} = \frac{f(\mathbf{x}^{+})}{f(\mathbf{x}^{*})}$$

where $f(x)$ is the objective function, x^{+} is the best-found solution during the optimization run, and x^{*} is the optimal solution. All results in this thesis are presented with their respective error. The results presented are the average of 5000 optimization runs for each method in each experiment.

6.1 Function Specific Learning

In this section, I will present the results of the tests designed to test the function-specific learning capabilities of the method.

6.1.1 $\alpha_{\text{CARLBO MM}}$ Tested on Multimodal Functions

Table 6.1 shows that the version of $\alpha_{\text{CARLBO MM}}$ (Sampling) is the best performing method, averaging around 96% of the max. $\alpha_{\text{MetaBO MM}}$ (Sampling) and EIpu performs slightly worse, at around 94%. We also observe that the argmax versions of the learned AFs underperform compared to their sampling counterparts, although $\alpha_{\text{CARLBO MM}}$ (Argmax) performs at around the level of EI-cool. We also see that $\alpha_{\text{CARLBO MM}}$ (Argmax) has a significantly longer length compared to the other methods.

Method	**Part of max**	**Length**
Random	0.8374 ± 0.0016	17.81 ± 0.05
EI	0.9338 ± 0.0013	17.67 ± 0.06
EIpu	0.9437 ± 0.0011	21.26 ± 0.07
EI-cool	0.9363 ± 0.0012	20.21 ± 0.06
$\alpha_{\text{CARLBO MM}}$ (Argmax)	0.9368 ± 0.0013	26.21 ± 0.08
$\alpha_{\text{MetaBO MM}}$ (Argmax)	0.8904 ± 0.0017	19.13 ± 0.10
$\alpha_{\text{CARLBO MM}}$ (Sampling)	$\mathbf{0.9593 \pm 0.0010}$	21.16 ± 0.05
$\alpha_{\text{MetaBO MM}}$ (Sampling)	0.9429 ± 0.0011	17.94 ± 0.05

Table 6.1: Results of $\alpha_{\text{CARLBO MM}}$ tested on 2-dimensional multimodal functions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

6.1.2 $\alpha_{\text{CARLBO GP}}$ Tested on Goldstein-Price Functions

By looking at table 6.2, we can observe that $\alpha_{\text{CARLBO GP}}$ (Sampling) performs the best, beating every benchmark other than $\alpha_{\text{MetaBO GP}}$ (Sampling) by over 11%. $\alpha_{\text{CARLBO GP}}$ (Argmax) performs closer to the level of the benchmarks and significantly worse than the sampling version. We also observe a large spread in the length of the average episodes, ranging from 17.99 in the case of the random agent, to 32.48 in the case of EI-cool. It is interesting to note that while there is a large difference in lengths between these two methods, their performance is very similar.

Method	Part of max	Length
Random	0.7652 ± 0.0013	17.99 ± 0.05
EI	0.7977 ± 0.0014	19.27 ± 0.08
EIpu	0.7827 ± 0.0014	30.91 ± 0.25
EI-cool	0.7788 ± 0.0014	32.48 ± 0.27
$\alpha_{\text{CARLBO GP}}$ (Argmax)	0.7923 ± 0.0020	28.62 ± 0.15
$\alpha_{\text{MetaBO GP}}$ (Argmax)	0.7834 ± 0.0020	19.22 ± 0.09
$\alpha_{\text{CARLBO GP}}$ (Sampling)	**0.9153 ± 0.0010**	21.90 ± 0.05
$\alpha_{\text{MetaBO GP}}$ (Sampling)	0.8965 ± 0.0013	20.72 ± 0.07

Table 6.2: Results of $\alpha_{\text{CARLBO GP}}$ tested on Goldstein-Price functions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

6.2 Generalization Across Different Dimensions

In this section, I present the results of the experiments designed to test how well CARLBO generalizes to functions with a different amount of dimensions than it was trained on.

6.2.1 $\alpha_{\text{CARLBO MM}}$ Tested on 3-dimensional Multimodal Functions With Two Cost Dimensions

When only considering the cost of sampling for two of the dimensions in the case of optimizing 3-dimensional multimodal functions, table 6.3 can see that α_{MM} (Sampling) performs the best, albeit only barely beating EI, with EI-cool not far behind. α_{MM} (Argmax) is the method with the highest average length, sampling about 50% more times than its sampling counterpart.

6.2.2 $\alpha_{\text{CARLBO MM}}$ Tested on 3-dimensional Multimodal Functions With Three Cost Dimensions

When moving to 3-dimensional multimodal functions where all three dimensions have a cost associated with them, table 6.4 shows that EI is the best-performing model. α_{MM} (Sampling), the best of the two learned AFs using CARLBO, is beaten by all the non-RL based benchmarks except for random. It is also worth noting that once again, α_{MM} (Argmax) has the highest average length, sampling about 36% more times than the method with the second highest average length.

Method	Part of max	Length
Random	0.6514 ± 0.0023	18.01 ± 0.05
EI	0.8043 ± 0.0022	19.02 ± 0.09
EIpu	0.7553 ± 0.0030	32.52 ± 0.13
EI-cool	0.7902 ± 0.0025	29.50 ± 0.12
$\alpha_{\text{CARLBO MM}}$ (Argmax)	0.7787 ± 0.0028	35.16 ± 0.11
$\alpha_{\text{MetaBO MM}}$ (Argmax)	0.7367 ± 0.0024	21.59 ± 0.13
$\alpha_{\text{CARLBO MM}}$ (Sampling)	**0.8092 ± 0.0021**	23.34 ± 0.06
$\alpha_{\text{MetaBO MM}}$ (Sampling)	0.7840 ± 0.0021	18.25 ± 0.05

Table 6.3: Results of $\alpha_{\text{CARLBO MM}}$ tested on 3-dimensional multimodal functions with evaluation cost in two dimensions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

Method	Part of max	Length
Random	0.6167 ± 0.0024	12.95 ± 0.03
EI	**0.7820 ± 0.0024**	13.84 ± 0.05
EIpu	0.7600 ± 0.0028	22.62 ± 0.08
EI-cool	0.7793 ± 0.0026	20.73 ± 0.08
$\alpha_{\text{CARLBO MM}}$ (Argmax)	0.6983 ± 0.0036	30.36 ± 0.11
$\alpha_{\text{MetaBO MM}}$ (Argmax)	0.7365 ± 0.0023	14.73 ± 0.07
$\alpha_{\text{CARLBO MM}}$ (Sampling)	0.7566 ± 0.0024	16.85 ± 0.04
$\alpha_{\text{MetaBO MM}}$ (Sampling)	0.7432 ± 0.0023	13.22 ± 0.04

Table 6.4: Results of $\alpha_{\text{CARLBO MM}}$ tested on 3-dimensional multimodal functions with evaluation cost in three dimensions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

6.3 Generalization to Different Function Types

In this section, I will present the results from the experiment that looks to assess the performance of the proposed model when used on function types that are different compared to the one it was trained on.

6.3.1 $\alpha_{\text{CARLBO MM}}$ Tested on 2-dimensional Noisy Convex Functions

When testing α_{MM} on 2-dimensional noisy convex functions, which exhibit similar properties, 6.5 shows us that EI-cool and EI perform equally well, both being only slightly ahead of EIpu. Both the sampling version and the argmax version of α_{MM} fall behind, with α_{MM} (Sampling) being the closest at about 0.5% behind. Like earlier, we observe that α_{MM} (Argmax) samples about 50% more times than the method with the second highest average length.

Method	Part of max	Length
Random	0.9209 ± 0.0008	17.43 ± 0.04
EI	$\mathbf{0.9709 \pm 0.0004}$	19.74 ± 0.13
EIpu	0.9698 ± 0.0004	19.79 ± 0.13
EI-cool	$\mathbf{0.9709 \pm 0.0004}$	20.01 ± 0.12
$\alpha_{\text{CARLBO MM}}$ (Argmax)	0.9438 ± 0.0007	33.63 ± 0.17
$\alpha_{\text{MetaBO MM}}$ (Argmax)	0.9633 ± 0.0005	20.84 ± 0.16
$\alpha_{\text{CARLBO MM}}$ (Sampling)	0.9669 ± 0.0004	21.62 ± 0.07
$\alpha_{\text{MetaBO MM}}$ (Sampling)	0.9622 ± 0.0005	18.47 ± 0.06

Table 6.5: Results of $\alpha_{\text{CARLBO MM}}$ tested on 2-dimensional noisy convex functions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

6.3.2 $\alpha_{\text{CARLBO MM}}$ Tested on Goldstein-Price Functions

When we test α_{MM} on Goldstein-Price functions, which should be significantly different, we see that the best performing method is α_{MM} (Sampling). It is almost 4% better than the second best method, which is $\alpha_{MetaBOMM}$ (Sampling). As earlier, we observe that α_{MM} (Argmax) samples approximately 60% more per optimization run compared to the method with the second highest average length. It is also worth noting that α_{MM} (Argmax) performs worse than random.

Method	Part of max	Length
Random	0.7635 ± 0.0013	17.50 ± 0.05
EI	0.7970 ± 0.0014	18.53 ± 0.07
EIpu	0.7811 ± 0.0014	27.46 ± 0.23
EI-cool	0.7797 ± 0.0014	26.32 ± 0.21
$\alpha_{\text{CARLBO MM}}$ (Argmax)	0.6957 ± 0.0018	44.27 ± 0.16
$\alpha_{\text{MetaBO MM}}$ (Argmax)	0.8037 ± 0.0016	19.08 ± 0.07
$\alpha_{\text{CARLBO MM}}$ (Sampling)	$\mathbf{0.8418 \pm 0.0011}$	20.72 ± 0.05
$\alpha_{\text{MetaBO MM}}$ (Sampling)	0.8075 ± 0.0012	17.72 ± 0.05

Table 6.6: Results of $\alpha_{\text{CARLBO MM}}$ tested on Goldstein-Price functions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

6.3.3 $\alpha_{\text{CARLBO GP}}$ Tested on 2-dimensional Multimodal Functions

As the Goldstein-Price functions don't have much variation, due to the only difference between the functions being that we vary the rotation, I expected the AFs learned on this function to overfit. As table 6.7 shows, EIpu is the best performing model, with EI almost exactly as well. All non-RL benchmarks, with the exception of random, perform better than all the RL-based ones, with $\alpha_{\text{CARLBO GP}}$ being the best performing Reinforcement Learning (RL)-based method.

Method	Part of max	Length
Random	0.8425 ± 0.0016	18.85 ± 0.06
EI	0.9362 ± 0.0013	17.52 ± 0.05
EIpu	$\mathbf{0.9363 \pm 0.0012}$	22.18 ± 0.10
EI-cool	0.9264 ± 0.0014	19.83 ± 0.07
$\alpha_{\text{CARLBO GP}}$ (Argmax)	0.7423 ± 0.0031	23.47 ± 0.14
$\alpha_{\text{MetaBO GP}}$ (Argmax)	0.7128 ± 0.0030	20.88 ± 0.13
$\alpha_{\text{CARLBO GP}}$ (Sampling)	0.9024 ± 0.0016	22.85 ± 0.11
$\alpha_{\text{MetaBO GP}}$ (Sampling)	0.8930 ± 0.0017	20.40 ± 0.10

Table 6.7: Results of $\alpha_{\text{CARLBO GP}}$ tested on 2-dimensional multimodal functions. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

6.4 Generalization to Different Cost Functions

In this section, we will observe the results of the experiments designed to test how CARLBO handles optimization problems with a different underlying cost function than it was trained on. Table 6.8 shows us that the best performing model is α_{MM} (Sampling), with $\alpha_{MetaBOMM}$ (Sampling) and α_{MM} (Argmax) not far behind.

Method	**Part of max**	**Length**
Random	0.8592 ± 0.0015	23.53 ± 0.07
EI	0.9508 ± 0.0011	22.78 ± 0.07
EIpu	0.9483 ± 0.0011	25.22 ± 0.07
EI-cool	0.9499 ± 0.0011	23.24 ± 0.07
$\alpha_{\text{CARLBO MM}}$ (Argmax)	0.9523 ± 0.0011	26.74 ± 0.08
$\alpha_{\text{MetaBO MM}}$ (Argmax)	0.8900 ± 0.0017	25.11 ± 0.08
$\boldsymbol{\alpha}_{\textbf{CARLBO MM}}$ **(Sampling)**	$\mathbf{0.9658 \pm 0.0009}$	24.40 ± 0.07
$\alpha_{\text{MetaBO MM}}$ (Sampling)	0.9538 ± 0.0010	23.20 ± 0.07

Table 6.8: Results of $\alpha_{\text{CARLBO MM}}$ tested on multimodal functions with a different underlying cost function than there were trained on. Part of max is how close the method got to the maxima of the function. All values are presented with error and is the average across 5000 optimization runs.

Chapter 7

Discussion

In this chapter, I will first discuss the meaning of the results observed in chapter 6, and attempt to offer insights into the significance of these results. I will then discuss some of the limitations of this study as well as some interesting potential areas for future research. Lastly, I conclude the thesis.

7.1 Main findings

In this section, I will first address the three research questions outlined in Section 1.2. Following this, I will explore the significant performance differences observed between the sampling and argmax versions of CARLBO, providing an analysis of their implications.

7.1.1 Assessing if Evaluation Cost Information Increases Performance

My first research question is concerned with determining if allowing an AF learned using RL to have information about evaluation cost increases its performance. One of the main findings of this thesis is that the sampling version of CARLBO consistently outperforms the MetaBO method across all experiments, suggesting that information about the evaluation cost indeed leads to enhanced performance. As anticipated, CARLBO, having access to the same information as MetaBO plus the added benefit of evaluation cost, facilitates learning more sophisticated search strategies. By being guided by the evaluation cost in certain instances, the algorithm can prioritize more promising options, thereby optimizing the balance between exploration and exploitation, and ultimately resulting in more efficient and effective optimization. This point is further reinforced by the finding that, on average across 7 out of 8 experiments, the sampling version of my method fit in more samples. The sole exception was the experiment with a different underlying cost function. This indicates that CARLBO has learned to more frequently sample from less costly regions of the input space.

7.1.2 Comparing My Method to Cost-Aware Benchmarks

In the second research question, I seek to compare the performance of CARLBO to other acquisition functions made with the same types of functions in mind. In my case, that is EIpu and EI-cool. My experiments show that the method proposed in this thesis outperforms EIpu and EI-cooling in five of eight experiments.

The first instance in which CARLBO was outperformed occurred during the testing of $\alpha_{CARLBOGP}$ on 2-dimensional multimodal functions. This outcome was anticipated since $\alpha_{CARLBOGP}$ was hypothesized to overfit the Goldstein-Price functions due to their specific characteristics. Hence, this overfitting likely contributed to the model's underperformance. However, this may not necessarily be viewed as an issue. The primary reasoning for training an acquisition function on similar nature functions to the Goldstein-Price one lies in the expectation that the objective functions the learned AF will later be applied on possess similar characteristics. In such cases, overfitting is less of a concern, and perhaps even an advantage.

The second one was $\alpha_{CARLBOMM}$ tested on noisy convex functions. In this case, CARLBO was beaten by both EIpu and EL-cool, albeit not by a large margin. Why this happens isn't entirely clear, but one likely reason is that $\alpha_{CARLBOMM}$ was trained on functions where normally at least one local maxima, with a value close to the value of the global maxima, exists in a part of the input space where evaluations are cheap. This is not the case for the noisy convex functions, where a significant amount of the time, all points with a value close to the global maxima will be expensive to evaluate.

The third and last experiment where CARLBO was beaten by the cost-aware benchmarks was $\alpha_{CARLBOMM}$ tested on 3-dimensional multimodal functions with three cost dimensions. An interesting note here is that the best performing model was EI, which doesn't consider cost at all. This perhaps suggests that due to the potentially high evaluation costs for a lot of the points x_i in the input space, most cost-aware methods considered in this thesis, my own included, take cost too much into account. Doing a lot of evaluations isn't helpful if you never start exploiting the knowledge. This hypothesis explaining $\alpha_{CARLBOMM}$'s underperformance is further supported by the observation that EI-cool, which significantly outperformed my method and was almost on par with EI, possesses a cooling component. This component gradually reduces the significance of evaluation cost as the optimization process progresses.

When benchmarked against the other cost-aware methods, CARLBO comes out on top in five out of eight instances. This implies that it is highly competitive, especially on functions similar to those it was trained on. However, it should be noted that these findings are based on a limited set of tests. To reach a more definitive conclusion, a broader and more diverse range of tests would be beneficial. This could potentially include a wider variety of functions and cost structures, which would help validate the robustness and adaptability of CARLBO.

7.1.3 Testing if the Learned Aquisition Function (AF) Generalizes Well

CARLBO has the highest performance in 3 of 6 experiments designed to test the method's capabilities for generalization. In the previous section, I speculated on the reasons why it might have underperformed in certain scenarios. Based on that, the data suggest that the AFs learned using CARLBO is capable of generalizing to different scenarios, outperforming the benchmarks when tasked to optimize 3 scenarios it had never seen before.

Generalization in machine learning typically relies on training with a diverse set of data. In my case, each learned AF was only trained on a specific type of function. I speculate that if CARLBO was used to train on multiple types of objective functions at once, such as both multimodal functions and noisy convex functions, we could have observed even better generalization capabilities.

7.1.4 Sampling versus Argmax

The results unambiguously demonstrate that the sampling versions of the learned AFs, for both CARLBO and MetaBO, consistently outperformed their argmax counterparts in almost every experiment. This suggests that the learned AFs considerably benefit from guided random exploration within the input space.

In RL, a policy's current decisions are guided by the assumption that the same policy will be used to decide future actions for the remainder of the episode. As a consequence, if a policy relies heavily on random exploration during training, its performance may well depend on the continuation of such exploration. This implies that the performance of the learned AF could hinge on a significant degree of random exploration.

Such dependence is far from ideal in an AF, as it can lead to considerable issues, particularly given the observed underperformance of the argmax versions. However, there may be potential strategies to address this issue, which will be outlined in Section **??**.

A distinctive pattern we observe is that the argmax version of CARLBO makes significantly more evaluations compared to any other method. This could suggest a situation where, due to its deterministic nature, it repeatedly samples the same query point x. If the evaluation cost for this point is low, the state doesn't change considerably as no new information is gained and the learned AF continues to query this identical point until the optimization run concludes. This can be perceived as a potential shortcoming of a deterministic approach.

However, one possible solution to this issue could be to employ a more refined mechanism for choosing the set of input points §. This could help avoid excessive repetition and could be achieved by opting for a non-static set, such as the one suggested by MetaBO. This solution is further backed up by the fact that MetaBO reported no such issues.

7.2 Limitations and Future Work

While this thesis delivers some valuable insights, it is not without its limitations that need to be taken into account when interpreting the results. In this section, I will discuss the most pressing amongst these limitations, all of which are areas that should be explored as future work.

7.2.1 Model Diversity

Firstly, I only trained two models. The limited number of models restricts the breadth of the results and doesn't allow for a firm conclusion about the overall capabilities of CARLBO to be made. With more models, I could better analyze how the training method performs across different kinds of scenarios, both in terms of objective functions and underlying cost functions. This limitation suggests the need for more comprehensive experimentation in future work.

7.2.2 Function Type

The second limitation is related to the diversity of the function types on which the models were trained. Each model was only trained on one particular function type, a choice that can significantly impact the model's ability to generalize to unseen scenarios. While this study demonstrated some promising results in terms of generalization, the training on a more diverse range of function types could potentially yield even better results.

7.2.3 Training of Models

Thirdly, the training of the models may have been insufficient. The reliance of CARLBO on random sampling to perform well might indicate that the models have not fully converged in a satisfying way, thus leaving some potential performance on the table. Either longer training times, or decreasing the entropy coefficient that hinders the policy from learning sharp distributions throughout training, could potentially reduce this reliance and improve the overall performance of the method, especially for the argmax version of the learned AFs. This is especially crucial as the argmax version is the version that is interesting for real life use.

7.2.4 Input Points Creation

The fourth limitation concerns to the creation of the set of input points $\mathcal{X}$. The method used in this study to generate $\mathcal{X}$ is not optimal and could have influenced the final results. In scenarios where the models sampled the same point excessively, a more dynamic or sophisticated method for choosing the set of input points might improve performance.

7.2.5 Lack of Testing on Real-World Problems

Another key limitation of this study is that the experiments were conducted using synthetic functions rather than real-world problems. Although synthetic functions are widely used when assessing performance in BO research, they might not adequately capture the complexities and nuances of real-world problems. This is especially apparent in my case, where all the functions considered had no noise in the evaluation, something that is very uncommon for real-world problems.

7.2.6 Cost Function Approximation

The underlying cost functions I have used in this thesis in order to train and test my models have been completely known, and I have provided the learned AFs with complete knowledge of the exact cost of sampling a given point. However, in practice, this is an unreasonable assumption, as we rarely know exactly the cost of sampling any given point in the input space. Swersky et al. (2013) suggested that the underlying cost functions can also be modeled using a GP. This would in turn add uncertainty to the estimates, which would need to be reflected in the information provided to the learned AF.

While these limitations are important to acknowledge, they do not undermine the significance of the findings made in this thesis but rather suggest areas to improve upon in future work, in order to make a more robust evaluation of the method's potential performance.

7.3 Conclusion

In this thesis, I proposed a novel method for training acquisition functions for Bayesian optimization using RL. This approach showed promising results, outperforming a wide range of benchmarks in several experimental scenarios. However, the results also highlighted a need for further investigation and refinement of the method, with the potential for significant improvements.

I found that CARLBO displayed a high capability to generalize, performing well on several different types of objective functions, across functions with different dimensionality and with different underlying cost functions. However, the experiments also revealed areas for improvement, particularly in relation to the model's reliance on random sampling.

I also pointed out several limitations that can be approved upon in future work, highlighting possible areas of improvement, including diversifying the training data, improving the training loop in order to be less reliant on random sampling, and creating a better input point set $\mathcal{X}$.

Overall, the work presented in this thesis offers a contribution to the domain of cost-aware Bayesian Optimization. While the results presented show promise, they underline the need for future research, which is further facilitated through the development of a framework for training cost-aware Aquisition Function that was presented in this thesis.

Bibliography

Mikhail Belyaev, Evgeny Burnaev, and Yermek Kapushev. Exact inference for gaussian process regression in case of big data with the cartesian product structure, 2014.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*, chapter 1.1. Cambridge University Press, 2004.

Eric Brochu, Matthew W. Hoffman, and Nando de Freitas. Portfolio allocation for bayesian optimization. *CoRR*, abs/1009.5419, 2010.
URL: http://arxiv.org/abs/1009.5419.

Kiran Chhatre, Sidney Feygin, Colin Sheppard, and Rashid Waraich. Parallel bayesian optimization of agent-based transportation simulation, 2022.

David Duvenaud. *Automatic Model Construction with Gaussian Processes.* PhD thesis, University of Toronto, 2014.
URL: https://www.cs.toronto.edu/~duvenaud/thesis.pdf.

David Eriksson and Martin Jankowiak. High-dimensional bayesian optimization with sparse axis-aligned subspaces, 2021.

FIDE. Laws of chess, 2021.
URL: https://handbook.fide.com/chapter/E012018. Accessed: 2023-07-30.

Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in Neural Information Processing Systems*, 2018.

Roman Garnett. *Bayesian Optimization.* Cambridge University Press, 2023.

John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.
URL: `http://jmlr.org/papers/v23/21-1342.html`.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995. doi: 10.1109/ICNN.1995.488968.

Eric Hans Lee, Valerio Perrone, Cedric Archambeau, and Matthias Seeger. Cost-aware bayesian optimization, 2020.

OpenAI. Openai five. OpenAI Blog, 2018.
URL: `https://openai.com/research/openai-five`.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning.* The MIT Press, 2006. ISBN 0-262-18253-X.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016. doi: 10.1109/JPROC.2015.2494218.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012.

S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Retrieved July 20, 2023, from `https://www.sfu.ca/~ssurjano/goldpr.html`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 2018.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.
URL: `https://proceedings.neurips.cc/paper_files/paper/2013/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf`.

Michael Volpp, Lukas P. Fröhlich, Kirsten Fischer, Andreas Doerr, Stefan Falkner, Frank Hutter, and Christian Daniel. Meta-learning acquisition functions for transfer learning in bayesian optimization, 2020.

Andrew Gordon Wilson and Ryan Prescott Adams. Gaussian process kernels for pattern discovery and extrapolation, 2013.

Martin Wistuba, Nathaniel Schilling, and Lars Schmidt-Thieme. Scalable gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, 107:43–78, 2018. doi: 10.1007/s10994-017-5684-y.