Hyperparameter Optimization of Opaque Models for Autonomous Vehicle Algorithms

by

Elaheh Ahmadi

B.S Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

© Massachusetts Institute of Technology 2021. All rights reserved.
Hyperparameter Optimization of Opaque Models for Autonomous Vehicle Algorithms

by

Elaheh Ahmadi

Submitted to the Department of Electrical Engineering and Computer Science on September, 2021, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Algorithms usually consist of many hyperparameters that need to be tuned to perform efficiently. It may be possible to tune a handful of parameters manually for simple algorithms however as the algorithm becomes more complex the number of hyperparameters also increases which makes finding the optimal hyperparameters more difficult. As a result, automating the parameter tuning would be of great interest in many different applications by reducing manual labor while increasing the performance of the algorithm. In this research, we focused on automating the process of hyperparameter selection for any opaque model to enable fully automated learning. We surveyed different hyperparameter optimization algorithms, selected the most efficient ones in different scenarios, and developed a framework that can be easily utilized by different users. We tested our algorithm and framework on NVIDIA’s localization algorithm developed for Autonomous Vehicles. Additionally, we performed hyperparameter optimization on different regression algorithms on the abalone [7] dataset to have another thorough comparison of the different optimization algorithm.

Thesis Supervisor: Samuel Madden
Title: Professor

Thesis Supervisor: Amir Akbarzadeh
Title: Engineer at NVIDIA
Acknowledgments

I want to take this opportunity to thank the many people who supported me in my journey at MIT and this thesis. From my amazing mentors and advisors at MIT and NVIDIA, to my friends and family. I'm extremely grateful to Amir Akbarzadeh and Ivana Stojanovic, who guided me during this research while we were adjusting to our new life during covid. I would also like to extend my deepest gratitude to Prof. Samuel Madden, who was my advisor for this thesis project. I'm also extremely grateful to Prof. Daniela Rus who welcomed me in her lab and was a caring mentor and helped me grow as a researcher. I also want to extend my thanks to my colleagues Alexander Amini and Qingyang Xu at Daniela Rus's lab whom I worked with on another project, Identifying and Mitigating the Bias in Predicting Drug Development Outcomes.

While I am extremely grateful to all of those who directly helped me to make this thesis a success, I want to extend my thanks to Pouya Bashivan, Jonas Kubilius, Xavier Boix, and Luca Carlone, my research mentors during my undergraduate time at MIT, without their guidance and mentorship I wouldn’t have become the person I am today. I’d also like to extend my gratitude to my friends at MIT and beyond who were my family and home away from home, Elham Banaie, Mehrangiz Banaie, Alexa Jan, Vibha Agarwal, Saba Nejad, Hengameh Bagherian, Sara Abdali, Shahin Roozkhosh, Aria Rezaei, Aria Halavari, Khadijah Sheikhan, Mostafa Khezri, Fatemeh Zebarjad, Donna Hakimian, Arman Rezaei, my lovely iHouse family, and my Tau Epsilon Phi sisters and brothers.

Additionally, I cannot begin to express my thanks to my parents Leila and Abdollah, and my lovely sister, Elnaz, who have been patient and supportive every day since I left my hometown Iran. I want to use this opportunity to say that I love you and I cannot wait to be reunited with you. I am also grateful for the support of my friends from home, Maryam Riazi and Amineh Ahmadinejad, and the support of my extended family.

In the end, I hope I can return all of your efforts and guidance by making a positive
change in this world. I believe my experience during this thesis helped to get one step closer to this goal.

Thank you I hope you enjoy reading this thesis.
Contents

1 Introduction .......................................................... 13
   1.1 Problem Statement ............................................. 14
   1.2 Our Approach ..................................................... 15

2 HPO Framework ...................................................... 17
   2.1 Design details ................................................... 18
       2.1.1 Optimizer .................................................. 19
       2.1.2 Opaque Model ............................................. 20
       2.1.3 Optimization Parameters ................................. 20
       2.1.4 Logging .................................................... 21

3 Background .......................................................... 23
   3.1 Search Based Optimizations ................................. 23
       3.1.1 Random Search .......................................... 24
       3.1.2 Grid Search ............................................... 24
   3.2 Population Based Optimizations ............................. 27
       3.2.1 Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) 27
   3.3 Bayesian Optimization ........................................ 28
       3.3.1 Probabilistic Surrogate Model .......................... 29
       3.3.2 Acquisition Functions ................................... 29
       3.3.3 BayesOpt .................................................. 30
   3.4 Multi-fidelity optimization ................................. 31
   3.5 Comparison of Algorithms .................................... 31
List of Figures

2-1 HPO Algorithm system diagram. ........................................... 18

4-1 Gridsearch optimization with BFS searching, the entire space is not
explored as early stopping was enforced. ................................. 35
4-2 Bayesian optimization with expected improvement acquisition function. 36
4-3 CMAES optimization. ......................................................... 37
4-4 KPI of segments in our dataset. m0, m1, and m2 are the name of three
main dataset we worked with. The y axis shows the KPI value for each
segment and the x axis shows the location of the segment with respect
to other segments. ................................................................. 39
4-5 Distribution of KPI values for the segments in our dataset. ........ 41
4-6 Result of running optimization on the batch chosen by clustering based
on segment-KPI distribution. The left figure shows the segments before
optimization and the right is the result after optimization. ........... 41
4-7 Clustering based on KPI value. ............................................... 43
## List of Tables

4.1 Comparison of performance for Bayesian, CMAES, and Grid search optimization for HPO ................................................. 34

4.2 Performance of picking a batch of segment via clustering based on segment-KPI distribution ............................................ 42

4.3 Performance of picking a batch of segment based on their KPI value ................................................................. 42

5.1 HPO for Multi-layer Perceptron Regressor on the Abalone dataset. Hyper-parameters are (hidden layer sizes, activation, solver, alpha, batch size, learning rate). The default parameters for MLPR were ((50), relu, adam, 0.0001, 30, constant) with score = 0.61 .... 46

5.2 Linear Regression on the Abalone Dataset ................................................. 47

5.3 HPO for Stochastic Gradient Descent Regression on the Abalone dataset. Hyper-parameters are (loss, penalty, alpha, l1 ratio, fit intercept, epsilon, learning_rate). Default parameters are (squared_loss, l2, 0.5, True, 0.1, invscaling) with score = 0.517 ............... 47

5.4 HPO for Elastic Net on the Abalone dataset. Hyperparameters are (alpha, l1 ratio, fit intercept, normalize). With default parameters (1, 0.5, True, False) with score = 0.336 ................................. 48

5.5 HPO for Bayesian Ridge Regression on the Abalone dataset. Hyperparameters are ($\alpha_1$, $\alpha_2$, $\lambda_1$, $\lambda_2$, fit intercept, normalize). With default parameters (1e-6, 1e-6, 1e-6, 1e-6 True, False) with score = 0.522 .... 49
5.6 HPO for Gradient Boosting Regression on the Abalone dataset. Hyperparameters are (loss function, learning rate, number of boosting stages, sub sample, criterion, max depth). With default parameters (ls, 0.1, 100, 1, friedman_mse, 3) with score = 0.56

5.7 HPO for Epsilon-Support Vector Regression on the Abalone dataset. Hyperparameters are (kernel, degree, C, gamma, coef0, epsilon). With default parameters (rbf, 3, 1.0, scale, 0, 0.1) with score = 0.54
Chapter 1

Introduction

Hyperparameters in a problem are any parameters that control the algorithm. For instance, the number of layers in a neural network (NN) and the number of neurons in each layer are hyperparameters. Therefore, choosing an optimal hyperparameter set is crucial in any problem as it can improve the quality of the algorithm. Hyperparameter optimization became of wide interest when machine learning algorithms became more popular and complicated. Researchers have tried different methods to find the optimal hyperparameters in any algorithm. In some cases, one can guess or easily estimate the hyper-parameters. For instance, if there is only one hyperparameter with a small range one can find the optimal value in a brute force matter. However, in a more complicated situation where instead of a single hyper-parameter, one is dealing with hundreds or even thousands of hyperparameters, the brute force method is out of the question. In this scenario, one needs to do a tedious calculation and manual labor and still, there is no guarantee that the set of hyper-parameters are optimal. As a result, designing an algorithm that can automate and optimize the hyper-parameter selection process became of great interest.

The performance of some algorithms can impact our lives directly. In the case of autonomous vehicles, we can avoid unfortunate incidents by improving the performance of localization or object detection. Mohamed et al. [19] provides a thorough survey of some of the algorithms developed in this field. For instance, in the task of localization, the pipeline usually consists of processing the input data from sensors
and radars, detecting objects in the environment, and ultimately estimating the location of the vehicle. There have been numerous works to perfect each segment of this complicated task [3]. If we look closely at each of these algorithms we observe that they all consist of many hyperparameters that need to be optimized.

AutoML [8] was proposed as a way of automating the entire learning pipeline; from training the model to optimizing it. One key part in AutoML [8] is automated Hyper-parameter Optimization (HPO). Many different algorithms target HPO. In most cases, the model that one aims to optimize should be treated as an opaque model. Opaque Model is any model that the user only has access to the output given a known input, and has no information regarding the structure of the model itself. In the following chapters, we will cover a thorough review of the current HPO algorithms that are widely used by experts.

1.1 Problem Statement

We are given an opaque model $f$. Our access to $f$ is only through an evaluation output given an input dataset $D$ and hyper-parameter set values $\Theta = \theta_1, ..., \theta_N$ such that $l = f(D, \Theta)$, where $l$ is the model evaluation. The problem we want to tackle is to find an algorithm that finds the optimal value of the hyper-parameters $P_{\text{optimal}}$ in the most efficient manner given some constraints.

There are three main constraints. First, we assume we have an initial best guess for each parameter we call our default parameters set, $\Theta_0 = \theta_{01}, ..., \theta_{0N}$. Second, we assume the optimal value lies within the range of the default parameters set, $R = r_1, ..., r_N$, where $r_i = (r_{i\text{lower}}, r_{i\text{upper}})$, and $r_{i\text{lower}}$ and $r_{i\text{upper}}$ are the lower bound and upper bound for parameter $\theta_i$ respectively. We also assume that $\theta_{0i}$ is within the given range, for $i \in [1, N]$. Third, we assume that we have limited resources, for instance the search for the optimal parameters set cannot take longer than $T_{\text{max}}$.

Finally we want to find $\Theta_{\text{optimal}} = \arg \min_{\theta} f(D, \Theta)$ such that $\theta_{i\text{optimal}} \in r_i$ and we utilized a limited amount of our resources. As a result, we want to find an algorithm $H$ such that $P_{\text{optimal}} = H(f, D, \Theta, R, \Theta_0, T_{\text{max}})$. Following chapters will compare
different options for $H$ and their accuracy and efficiency.

### 1.2 Our Approach

In order to tackle this problem, we divided the problem into three segments.

1. Design and implement a hyper-parameter optimization (HPO) framework for any opaque model that enables us to run HPO on any model and reduce manual labor.

2. Implement different hyper-parameter optimization algorithms and plug-in available open-source libraries into the framework.

3. Survey possible solutions for reducing optimization time.

The first segment will allow us to have a framework where we can run different experiments and test our ideas on any model. Additionally, in our design, we considered a user-friendly interface such that the framework can be utilized by other users to optimize their algorithms. Chapter 2 goes over the design details of this framework. Once the framework was set in place we surveyed different optimization algorithm that has been utilized in this domain and implemented them in our framework. We added Grid Search Optimization 3.1.2, Bayesian optimization 3.3, and Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) 3.2.1. Chapter 3 explains these optimization algorithms in detail along with random search and some other optimization methods. We then tested our framework on NVIDIA’s localization algorithm. From our experiments, Bayesian optimization and CMA-ES had almost similar and equally good performance on this model as we varied the number of parameters. While grid search in some cases may find better hyperparameter values, the computation time is much more than the other two optimizations. Chapter 4 compares different optimization algorithms extensively and some of the methods we utilized to improve the optimization speed and performance. Since NVIDIA’s localization algorithm is very complicated and includes many different hyperparameters, we also ran our framework on other models. We picked [7] dataset and optimized different regression algorithms.
Our result is however similar to chapter 4 in the sense that CMA-ES and Bayesian optimization have very similar performance. and regression models on abalone [7] dataset. In Chapter 5 we go over the optimization results. At the end in chapter 6 we discuss a summary of this thesis and its contribution.
Chapter 2

HPO Framework

In this chapter, we will explain in detail the HPO framework that we developed. We designed it with the goal of making it as user-friendly and efficient as possible. The framework was developed in python3. The HPO framework can be utilized to optimize any set of parameters (we will explain in detail what type of parameters were supported) for any opaque function such that the user can plug in their opaque model, specify the parameters they want to optimize, and then run the optimization pipeline to find the best set of parameters.

As mentioned in section 1.1 the problem that we are aiming to solve is as follows: Given a set of data and an opaque model, we want to find a set of hyper-parameters, \( \Theta = \theta_1, ..., \theta_N \), with ranges, \( R = r_1, ..., r_N \), such that the opaque model reaches a minimum cost or maximum productivity given some imposed constraints. Figure 2-1 shows the problem as a system diagram. Based on the problem we have we designed the framework such that it consists of three main components: Optimizer, OpaqueModel, and OptParams. As evident by their name, each component abstracts different parts of the HPO algorithm. In the following sections, we will go over our design detail and usage.
2.1 Design details

The HPO algorithm requires the user to specify the opaque model, \( f \), the default value for hyper-parameters they want to optimize, \( P_0 \), the ranges for each parameter, \( R \), and any constraint, \( C \), they want to apply to the optimization process. In return, the algorithm outputs the optimal value of hyperparameters, \( \Theta_{t*} = [\theta_{t*1}, \ldots, \theta_{t*N}] \), and minimum cost \( cost_{t*} \). As a result, three abstractions were designed to take care of the main parts of the algorithm. The optimization section is handled by \textit{OptimizationParameters}, where it utilizes an optimization algorithm to optimize an opaque function by searching in the hyper-parameter space. The optimization parameter is handled by \textit{OptParams} and provides an easy user interface (UI) to define the optimization parameters, change their ranges, define their types, etc. And finally, the opaque function is handled by the \textit{OpaqueModel}. Since different users may have different interfaces with their opaque functions, we designed a general abstraction where users can plug in their opaque functions and utilize the HPO framework.
2.1.1 Optimizer

As evident from the name, Optimizer is an abstraction for different optimization algorithms that can be utilized for HPO. This class has two functions that have to be implemented for the subclasses. \_\_init\_\_() and optimize().

\_\_init\_\_()

This function initializes the Optimizer and takes an instant of OpaqueModel class and optimization related setup as an input.

optimize()

This function must be implemented by all the Optimizer subclasses, and as evident by its name, is where the optimization is happening.

Subclasses

We implemented BayesOptimizer, CMAESOptimizer, and GridSearchOptimizer in our framework. For BayesOptimizer we utilizes Bayesian Optimization [24] library, for CMAESOptimizer we plugged in pyCMA [23] library, and for GridSearchOptimizer we implemented our own optimizer that uses BFS, DFS, and SmartDFS as it’s search option. You can learn more about the implementation and the theory behind each optimizer in chapter 3.

Optimizer Handler

Additionally, we implemented OptimizerHandler that initializes an optimizer based on what the user has specified. The Handler is what the users interact with to initialize the optimizer. This handler, similar to Optimizer, has two functions, \_\_init\_\_() and optimize() where the user can use the former to pass in their opaque model and their optimization setups and use the later to run the optimization.
2.1.2 Opaque Model

We designed *OpaqueModel* abstraction to create a general structure for what functionally an instance of *OpaqueModel* must have. This abstraction allows users to plug in their own opaque model and use HPO. When a user creates an *OpaqueModel* subclass they must implement `get_opaque_model_function(...)`, and `get_default_params(...)`.

`get_opaque_model_function(...)`

This is where the user plugs in their opaque model. In this function, you must return a function that takes in the hyper-parameters as an input and returns the loss value for minimization or an evaluation value for maximization.

`get_default_params(...)`

This function must return the default parameters of the hyperparameters user aims to optimize.

Opaque Model Handler

Additionally, we designed an *OpaqueModelHandler* for user interaction with an *OpaqueModel* instance.

2.1.3 Optimization Parameters

In different optimization problems, users might have different types of hyperparameters they aim to optimize. Let’s consider an example where we want to optimize the number of layers, number of neurons in each layer, and different activation functions in each layer. The number of layers and neurons must be integers and the activation functions must be chosen from a list of options. In this scenario, we have two different types of parameters that require different ways of being handled. In more complicated scenarios, we might have continuous parameters or some parameters that depend on each other. Therefore, we designed the *OptParams* and *OptParam* abstraction to handle different optimization parameters. *OptParams* handles a group of different
OptParam instances. Moreover, OptParam is a parent class of different types of optimization parameters. In our framework, we support integers, continuous, categorical, and Boolean parameters. In some cases, some parameters might depend on each other in some ways, for these scenarios designed our OptParam data structure such that some parameters are actively being optimized and some are passively being optimized through the active parameters.

2.1.4 Logging

One important part of our design was implementing logging during optimization. While what we are looking for is the optimal value, it is crucial for us to know how the optimizer searched the space and what were the other values. Moreover, in the case that our optimization was interrupted for any reason, by keeping a track of the explored area we can resume the optimization without having to perform the optimization from the beginning.

In the following chapter, we will explain each of the optimization algorithms in more detail.
Chapter 3

Background

In this chapter, we will go over different optimization algorithms that can be used for Hyper-Parameter Optimization (HPO). In all of these methods, we assume the model that we are optimizing its hyperparameters for is opaque and therefore we don’t have access to the details of the model and the optimization should not depend on it.

3.1 Search Based Optimizations

In search-based optimization, we are given a range for each hyperparameter, therefore we know the hyperparameter space. In this method, we assume the optimal hyperparameter values lie in the hyperparameter space that we are given. As a result, our problem is simplified into a search in a space where every option in the hyperparameter space is equally likely to be the optimal answer. In the algorithm below we will assume our parameter space is discrete and the maximum size of the space is $d^N$, with $d = \max_{i \in 1,...,N}(size(r_i))$. We also assume the time to evaluate the opaque model given a set of hyperparameter values is $T_f$. In the following section, you can find the description of different search-based algorithms with their pseudo-code and run time and efficiency calculation.
3.1.1 Random Search

One simple solution is random search[14]. In this algorithm we set a maximum number of iterations, $I_{max}$, we want to search the space. In every step, we randomly pick a point from the hyperparameter space, and evaluate the opaque model. We repeat until the maximum number of iterations is reached and then pick the hyperparameter values that resulted in the best performing opaque model. Algorithm 1 gives a pseudo-code of how random search can be implemented. [8]

Algorithm 1 Random Search for HPO

1: function RANDOMSEARCH($f, D, R, [\theta_1, \theta_2, \ldots, \theta_N], [r_1, r_2, \ldots, r_N]$)
2:     explored = []
3:     for length(explored) <= $I_{max}$ do
4:         $\Theta = []$
5:             for $\theta_i$ in $\theta_1, \theta_2, \ldots, \theta_N$ do
6:                 $\theta_i \leftarrow$ pick at random from $(r_{i_{lower}}, r_{i_{upper}})$
7:                 $\Theta$.append($\theta_i$)
8:             end for
9:             if $\Theta$ not in explored then
10:                cost $\leftarrow f(D, \Theta)$
11:                explored.append($\Theta$)
12:            end if
13:     end for
14:     return $\Theta$ with min cost from explored
15: end function

Random search run-time is $T_f * I_{max}$, which could be useful in problems where the hyperparameter space is small. In such cases, if we have an idea of what is the smallest achievable cost then we can impose early termination once the best cost is achieved. However, the chances of finding the optimal hyperparameter values get lower as the space grows, making this method less useful in the practical sense.

3.1.2 Grid Search

Grid search [26] is an algorithm that exhaustively explores all the possible options in the search space. The simplest method to finding the best hyperparameter values is grid search. In this approach, the algorithm requires the evaluation of the opaque
model on the entire hyperparameter space. While this algorithm is guaranteed to
return the best possible set of parameters (if the search is indeed on the entire hy-
perparameter space), it is not efficient for high dimensional hyperparameter spaces
as the search time increases exponentially with the dimension of the space. An ex-
ample of a grid search can be found in pseudo-code 2. This algorithm must use some
method for searching the space. One evident way to search the space is a random
search which was explained in 3.1.1. Below we will explain some other methods for
searching the hyperparameter space. In all of these methods, the computation time
is $O(d^N)$, which can be improved with early termination with the cost of finding a
sub-optimal solution.

### Algorithm 2 Grid Search for HPO

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>function GridSearch(f, D, search_function, R, [θ₁, θ₂, ..., θₙ], [r₁, ..., rₙ], Θ_default)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>explored = []</code></td>
</tr>
<tr>
<td>3</td>
<td><code>costs = []</code></td>
</tr>
<tr>
<td>4</td>
<td><code>Θ ← Θ_default</code></td>
</tr>
<tr>
<td>5</td>
<td><code>while space is not fully explored do</code></td>
</tr>
<tr>
<td>6</td>
<td><code>if Θ not in explored then</code></td>
</tr>
<tr>
<td>7</td>
<td><code>cost ← f(D, Θ)</code></td>
</tr>
<tr>
<td>8</td>
<td><code>explored.append(Θ)</code></td>
</tr>
<tr>
<td>9</td>
<td><code>costs.append(cost)</code></td>
</tr>
<tr>
<td>10</td>
<td><code>end if</code></td>
</tr>
<tr>
<td>11</td>
<td><code>Θ ← getNextParameter(Θ, R, explored)</code></td>
</tr>
<tr>
<td>12</td>
<td><code>end while</code></td>
</tr>
<tr>
<td>13</td>
<td><code>return Θ with min cost from explored</code></td>
</tr>
<tr>
<td>14</td>
<td><code>end function</code></td>
</tr>
</tbody>
</table>

### BFS

Breadth-first search (BFS) algorithm is utilized in finding a specific node in tree or
graph data structures. This algorithm starts from one node, explores that node and
adds all of its unexplored neighbors to a queue. Then repeats the same process for
the first item in the queue until the queue is empty or the specific node is found.
BFS can be used to search for the optimal hyperparameter values in grid search
optimization. Let’s assume we only have two parameters $θ₁$ and $θ₂$, with default
values (0, 0) respectively, and both parameters range from (-2, 2) and are integers. In
this scenario BFS starts from the default parameter (0, 0), explores the neighbors of this parameter that are (1, 0), (0, 1), (-1, 0), (0, -1). And continues by exploring the neighbors of (1, 0), avoiding the already explored parameters, until the entire space is searched.

**DFS**

Depth-first search algorithm is another search algorithm that is utilized in graph and tree structure data. This algorithm starts from one node finds all the neighbors of that node and explores the first unexplored neighbor, recursively continues the process until it reaches a node that does not have any unexplored neighbor. The process is continued until there is no more unexplored node or the target node is found. DFS, similar to BFS, can be utilized in grid search for HPO. Let’s use the simple example from above. In this scenario however the algorithm explores, (1,0) then (2,0) and (-1, 0) and (-2, 0). And then explores (0, 1), (0,2), (0, -1), and (0, -2) and so on until the entire hyper-parameter space is explored. One might wonder if the goal is to explore the entire space then there isn’t any benefit in using BFS, DFS, or even random search. And that is indeed true however if the user has some initial idea of where the optimal hyperparameter values stand in the space then using BFS might be preferred since early stopping can be enforced once a good enough answer is found. Additionally, we introduce another searching method where it doesn’t explore the entire space.

**Smart DFS**

Let us assume that our opaque function is such that optimization of hyperparameter values can be performed independently for each hyperparameter. In this case, we can modify the grid-search to consider this assumption. We introduce Smart DFS. In this algorithm, the search is performed very similarly to DFS. The algorithm searches along one dimension of the hyperparameters finds the parameter that minimizes the cost and the searches along the next dimension updating the last parameter to the optimal value found before. Going back to our simple example from before, the first
four steps are similar to DFS, and let us say the opaque function gives us the cost of 1, 2, 0, 3 for (1,0), (2,0), (-1, 0) and (-2, 0) respectively. Since (-1, 0) resulted in the minimum cost, the DFS now explores (-1, 1), (-1, 2), (-1, -1), (-1, -2). If the minimum cost is (-1, 0) the search is done and we found the optimal solution, otherwise we continue the search along the axis until either the entire space is explored or we found the optimal solution. The computation complexity of Smart DFS is similar to BFS and DFS in the worst-case scenario, however, on average it is faster if our initial assumption stands.

3.2 Population Based Optimizations

Population-based optimizations are one of the commonly used optimization algorithms for hyperparameter optimization. [16] [13] [4]. Here we will talk about a population-based algorithm that is being widely used for this purpose.

3.2.1 Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES)

Covariance Matrix Adaptation Evolutionary Strategy (CMAES) has been shown to have great performance for optimizing hyper-parameters of Deep Neural Networks[12]. CMAES is an evolutionary optimization algorithm that samples configuration from multivariate Gaussian distribution whose mean and covariance is updated based on the success of the population selected in each iteration [8]. CMA-ES can be applied in cases where we aim to optimize a large number of parameters up to a couple of hundreds. In the following chapter, you can find a comparison of this algorithm’s performance for optimizing hyper-parameters of NVIDIA’s localization algorithm.

Algorithm

Hansen N. et al.[22] introduced CMA-ES as an evolutionary global optimization method. This method creates a distribution over the hyperparameter space. In
every iteration, it samples a population from the parameter space. Then it evaluates these candidates by passing them to an Opaque function. Once the evaluation has been completed it passes the tuple of parameters and evaluation result into the parametric distribution model and updates its model by changing the mean and variance of the distribution model. In more detail, CMA-ES keeps track of a multi-variate normal distribution parameterized by $\theta$, $N(\theta, \mu, C)$, over the hyper-parameter space, $R^n$, where $\theta \in R^n$, $\mu$ is the mean and $C$ is the covariance of the normal distribution. In each iteration $k$, the algorithm generates $b$ samples from the normal distribution $\theta_i \sim N(\theta, \mu_k, C_k)$ for $i = 1, \ldots, b$. It evaluates the candidates on the opaque function and picks the top $\eta$ of the candidates to update the distribution. The algorithm continues until it either converges or has reached a maximum number of iterations. [21] [11]

**pycma**

The code for this algorithm is also available through multiple sources. In this thesis we utilized CMA-ES/pycma [23] library. Algorithm 3 shows a usage example of this library.

### Algorithm 3 pycma usage

```python
1: function CMAESOptimizer($f, D, R, [\theta_1, \theta_2, \ldots, \theta_N], lower\_bounds, upper\_bounds, \Theta_{default})$
2:     es = cma.CMAEvolutionStrategy($\Theta_{default}, \sigma$, {'bounds': [lower\_bounds, upper\_bounds]})
3:     es.optimize($f$)
4:     return es.result
5: end function
```

### 3.3 Bayesian Optimization

Bayesian optimization [5] is an iterative global optimization algorithm for expensive opaque functions. This algorithm consists of two key components, a probabilistic surrogate model for estimating an opaque model and an acquisition function that decides the next point to evaluate. This algorithm has been widely used to enable Auto-ML[8]. In this thesis we used In the following section, we will explain the task
of a probabilistic surrogate model and acquisition function.

### 3.3.1 Probabilistic Surrogate Model

The probabilistic surrogate model, $f(\theta)$ for $\theta \in \mathbb{R}^n$ the hyper-parameter space, keeps a prior distribution of what the opaque function may look like over the hyperparameter space given the hyperparameter values we have observed[5]. As a result, $f(\theta)$ provides us with the potential value of the opaque model for candidate $\theta$. At iteration $k$, we observe the value of $\theta_k$ on the opaque model and update the posterior distribution of the surrogate model [9]. In most cases, a Gaussian processes (GP) regression is employed as the surrogate model. A Gaussian processes regression is a Bayesian statistical method that can be utilized for modeling functions [6].

### 3.3.2 Acquisition Functions

The acquisition function is responsible for picking the candidate to explore at each iteration. Here we will explain the choices that are available in BayesOpt library [24] such as Expected Improvement (EI), Probability of Improvement (PI), and Upper Confidence Bound (UBC). Be aware that we assumed in these acquisition functions that our goal is to find the maximum value of the opaque function.

#### Probability of Improvement (PI)

This acquisition function, $\alpha_{PI}$, aims to maximize the probability of improvement over the current point with maximum function value, $f(\theta')$. As a result, in every step, the acquisition function finds the point with the highest probability of its value being greater than $f(\theta')$[2][28]. Equation 3.1 depicts the POI acquisition function.

$$\alpha_{PI}(\theta, f) = \arg\max(P(f(\theta) > f(\theta'))) = \Phi\left(\frac{\mu_k(\theta) - f(\theta')}{C(\theta)}\right) \quad (3.1)$$
Expected Improvement (EI)

The Expected Improvement acquisition function picks the candidate that has the highest expected improvement over the current maximum point, $f(\theta')$. Let’s define a utility function that returns 0 for all the points on the function $f(\theta)$ that are below the max and return the difference between the function value and the max everywhere else. Equation 3.2 depicts such utility function. Thus, the acquisition function can be defined as finding the point that maximizes the expected value of $u(\theta)$. Equation 3.3 defines this acquisition function. [2] [9]

$$u(\theta) = \max(0, f(\theta) - f(\theta'))$$

$$\alpha_{EI}(\theta, f) = \mathbb{E}[u(\theta)|\theta] = (\mu_k(\theta) - f(\theta')) \Phi\left(\frac{\mu_k(\theta) - f(\theta')}{C(\theta)}\right) + C(\theta) N\left(\frac{\mu_k(\theta) - f(\theta')}{C(\theta)}\right)$$

Upper Confidence Bound (UCB)

This acquisition function balances between exploring the hyperparameter space and exploiting it and $\beta$ is a trade-off parameter between the two. You can learn more about this acquisition function and more studies in Berk et al. [15].

$$\alpha_{UCB}(\theta, f; \beta) = \mu_k(\theta) + \beta \sqrt{C(\theta)}$$

3.3.3 BayesOpt

The code for this algorithm is also available through multiple sources. In this thesis we utilized BayesOpt [17] library. Algorithm 4 shows a usage example of this library.
Algorithm 4 BayesOpt usage

1: function BAYESOPTIMIZER($f, D, \text{param\_boundaries}, \Theta_{\text{default}}, n_{\text{iter}}, \text{acquisition\_func\_name}$)
2: bo = BayesianOptimization($f$, param_boundaries)
3: bo.probe($\Theta_{\text{default}}$)
4: bo.maximize($n_{\text{iter}}, \text{acquisition\_func\_name}$)
5: return bo.max
6: end function

3.4 Multi-fidelity optimization

Multi-fidelity optimization is a common technique to speed up manual tuning by using low-fidelity approximations of the actual loss function by probing the algorithm or hyper-parameter configuration on a smaller set of data or subset of features[8]. This algorithm is a great option when evaluating the opaque model is expensive. While it introduces a trade-off between optimization performance and run time, in practice the optimization error is outweighed by the speedup [8].

3.5 Comparison of Algorithms

In chapter 4 and 5 we provided a thorough comparison of the different optimization algorithms.
Chapter 4

HPO on Localization Algorithm

4.1 Localization Algorithm as Opaque Model

The main motivation of this thesis was to design and implement an algorithm that optimizes the hyperparameter values of NVIDIA’s localization algorithm. While we won’t get into the details of this algorithm for confidentiality reasons, we can talk about the high-level specification of this model. First of all, we treat this algorithm as an opaque model. We have access to some data that we can run the opaque model on and we know what are the parameters we want to optimize, the range of each parameter, and the default value for each parameter. This opaque model given a set of parameters $\Theta$ with their ranges $R$ and dataset $D$, will return a key performance indicator (KPI), $KPI$ that ranges from $(0 - 1)$. Air goal is to get $KPI$ to be as close as to 1 if not 1.

In this chapter, we will compare different optimization algorithms that we utilized to perform this optimization. Additionally, to improve the performance of the optimization we designed some algorithms to run the optimization on a subset of the dataset with the goal of increasing the optimization speed without loss of performance.
Table 4.1: Comparison of performance for Bayesian, CMAES, and Grid search optimization for HPO

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Default parameters</th>
<th>Default KPI</th>
<th>Optimal parameters</th>
<th>Optimal KPI</th>
<th>Time per step (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>(100.0, 5.0)</td>
<td>99.51</td>
<td>(600.0, 9.0)</td>
<td>99.73</td>
<td>79</td>
<td>1362</td>
</tr>
<tr>
<td>CMAES</td>
<td>(100.0, 5.0)</td>
<td>99.51</td>
<td>(250.0, 8.2)</td>
<td>99.67</td>
<td>93</td>
<td>88</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>(100.0, 5.0)</td>
<td>99.51</td>
<td>(222.8, 10.0)</td>
<td>99.64</td>
<td>92</td>
<td>106</td>
</tr>
</tbody>
</table>

4.2 HPO with full data

In this section, we will compare the performance of different optimization algorithms when we use the full data to calculate the KPI. In this experiment we focused on optimizing two parameters, \((p_1, p_2)\), with their default value \((100.0, 5.0)\). You can find the performance of each of the algorithms in table 4.1. As you can see Grid-search optimization finds the best hyperparameter. However considering the number of steps that it takes for this algorithm, 1362 with early stopping, we can easily conclude that CMAES and Bayesian optimization are better choices. Additionally, we can see that for these two parameters CMAES and Bayesian optimization’s performance are almost similar, with CMAES finding slightly better hyperparameter values faster. Moreover, by looking at figure 4-1 we observe that the area with the green circle is the area of interest (i.e. where the optimal hyper-parameter lies with high probability). If we look at the same area in both figure 4-2 and figure 4-3 we see that both of these algorithms also spend most of their exploring these areas after realizing that the area outside has a lower probability of containing the optimal hyperparameter values.

From these results, we can conclude that Bayesian optimization and CMAES are a better option than Gridsearch with BFS in terms of computation resource efficiency. The next challenge we tackled was, is there was a way to make the optimization
Figure 4-1: Gridsearch optimization with BFS searching, the entire space is not explored as early stopping was enforced.
Figure 4-2: Bayesian optimization with expected improvement acquisition function.
Figure 4-3: CMAES optimization.
process faster or more efficient. In the following section, we will cover some of the ideas we tried and their results.

4.3 Sub-data selection

Computing the KPI of our opaque model is the main bottleneck in the optimization as it takes a lot of computing power and time. Thus we decided to work with segments of our data instead of the entire data. Figure ?? depicts an example of KPI of different segments in our data. Using segments of the data instead of the full data allows us to run the optimization on a smaller dataset that represents the full dataset and reduce the computation time of KPI for different hyperparameter values.

We tried many different methods to improve the optimization performance. Some of the methods are as follows: optimizing on the hardest segment (i.e. the segment with the lowest KPI), optimizing on a batch of hard segments, different clustering methods to pick a subset of segments that represent the full data, and running multiple rounds of optimization and in each round change the segment we run the optimization on. In the following sections, we will go over some of these methods in more detail and study their performances.

4.3.1 Picking Hardest Segments

In this method, we find the hardest segment or in other words the segment with the lowest KPI and pick that segment for the optimization. This method might seem appealing since improving the worst-performing segment can improve the overall KPI, however in practice that didn’t seem to be the case. One instance where this method will undoubtedly fail is when the hyperparameter values we are optimizing is not the cause of that segment is poor performance. Our experiments also prove the same hypothesis that this method is not helpful.
Figure 4-4: KPI of segments in our dataset. m0, m1, and m2 are the name of three main dataset we worked with. The y axis shows the KPI value for each segment and the x axis shows the location of the segment with respect to other segments.
4.3.2 Clustering Segments

Instead of only picking the hardest segments, we decided to cluster the segments based on some criteria or labels. Then pick a batch of segments from different clusters such that the final selection of segments represents the full data fairly.

Clustering based on segment-KPI distribution

We propose a better way to pick a batch of segments for optimization. In this method, we find the distribution of segments over KPI values, and then we sample \( N_{batch} \) segments from that distribution and perform optimization over those segments. Figure 4-5 depicts an example of the distribution of the KPI for different segments. This method will ensure that the batch of segments we are using for optimization has the same underlying distribution as the entire data. After performing optimization using this method we observed that this method performs better than only picking difficult segments (Figure 4-6. However, once we ran the optimal hyperparameter values for the batch on the entire data we observed that the parameter that might improve performance for a batch of segments might not have the same performance on the entire data. We ran the HPO this time using the smartDFS 3.1.2 algorithm as our optimization, optimizing only a single parameter to see if this method is useful before jumping on optimizing a larger set of parameters. We repeated this experiment 3 times every time sampling from the distribution at random, table 4.2 shows the result. As you can see the algorithm could successfully find better hyperparameter values on the full data using the smaller batch 1 out of 3 times. In conclusion, while this method could find better hyperparameter values that default faster, we cannot rely on the result in most cases.

Clustering based on KPI value

One issue with the previous method was that since we are choosing our batch from the underlying distribution the chances of picking a segment with a KPI below 0.90 is quite low. As a result, we came up with the idea to cluster based on KPI values.
Figure 4-5: Distribution of KPI values for the segments in our dataset.

Figure 4-6: Result of running optimization on the batch chosen by clustering based on segment-KPI distribution. The left figure shows the segments before optimization and the right is the result after optimization.
Table 4.2: Performance of picking a batch of segment via clustering based on segment-KPI distribution

<table>
<thead>
<tr>
<th>Default KPI on segments</th>
<th>Best KPI on segments</th>
<th>Default KPI on entire data</th>
<th>KPI form optimal param on entire data</th>
<th>default parameter</th>
<th>Optimal parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.23</td>
<td>98.49</td>
<td>97.42</td>
<td>97.11</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>90.02</td>
<td>90.49</td>
<td>97.42</td>
<td>97.11</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>98.33</td>
<td>98.41</td>
<td>97.42</td>
<td>97.52</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.3: Performance of picking a batch of segment based on their KPI value

<table>
<thead>
<tr>
<th>Default KPI on segments</th>
<th>Best KPI on segments</th>
<th>Default KPI on entire data</th>
<th>KPI form optimal param on entire data</th>
<th>default parameter</th>
<th>Optimal parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>83.76</td>
<td>84.09</td>
<td>97.42</td>
<td>97.50</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>95.3</td>
<td>95.3</td>
<td>97.42</td>
<td>97.42</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>96.87</td>
<td>96.91</td>
<td>97.42</td>
<td>97.31</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 4-7 shows an example of how this method works. If $KPI_{min}$ is the minimum KPI among the segments and $KPI_{max}$ is the maximum, we divide the area into $n$ (equal to the number of segments we want to pick) sections based on the KPI value, such that the height of each section is $h = \frac{KPI_{max} - KPI_{min}}{n}$ theretfor the first section contains all the segments that their KPI lies in $[KPI_{min}, KPI_{min} + h)$ and so on. Now in order to select our batch of segments, we pick a segment at random from each section. If a section contains no segments we pick a segment from the neighbor section at random. We used the same setup as the previous section and ran the optimization 3 times. You can find the result in table 4.3. Based on these results, we can conclude the same thing as the previous section such that while this method might work for other datasets it is not promising in this scenario.
Figure 4-7: Clustering based on KPI value.
4.3.3 Multi-round Optimization

In addition to the methods we mentioned above, we also tried running multiple rounds of optimization and changing the segments in each round. However, the results were still not promising. We observed that in order to achieve good and reliable performance from sub-data selection, we need to run the optimization for multiple rounds on different segments and choose the best performing among them, however, we realized this method will end up taking as much time as if we just ran the optimization once of the full dataset.

4.4 Chapter Conclusion

We dedicated this chapter to the result of running HPO framework on NVIDIA’s localization algorithm and finding methods to improve the speed and performance of the optimization. While our results may not show a complete study of the different optimization algorithms, it shows that the framework is successfully working on this algorithm. Moreover, the framework was integrated into their pipeline and is being used to optimize the hyperparameters on a regular basis one the full data. We dedicated the next chapter solely to study the different optimization algorithms on some networks that are publicly available and examined how their performance changes as we increase the number of hyperparameters.
Chapter 5

HPO on Other Machine Learning Models

For this chapter, we used the Abalone UCI dataset [7] and ran the optimization on various Machine Learning algorithms to predict the age of abalone based on the features such as Sex, Length, etc. We utilized *scikit-learn*’s [25] open-source algorithms for this problem. Since the nature of this problem is a regression problem we used Multi-layer Perceptron Regressor (MLPR), Linear Regression, Stochastic Gradient Descent Regression, Elastic Net Regression, Bayesian Ridge Regression, and Gradient Boosting Regression. In the following sections we will compare the performance of these classifiers and how much our HPO could boost the performance. It is worth mentioning that the default values are set to the default values of each *scikit-learn* algorithm.

5.1 Comparing performance of HPO

In this section, we will go over the performance of different HPO algorithms. Since in most of these algorithms the hyper-parameters are not independent of each other we won’t use Smart DFS 3.1.2 and focus our study to Bayesian Optimization 3.3, Covariance Matrix Adaptation Evolutionary Strategy 3.2.1, and Grid Search with BFS 3.1.2.
Table 5.1: HPO for Multi-layer Perceptron Regressor on the Abalone dataset. Hyperparameters are (hidden layer sizes, activation, solver, alpha, batch size, learning rate). The default parameters for MLPR were ((50), relu, adam, 0.0001, 30, constant) with score = 0.61

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>[(70), relu, adam, 0.0001, 30, adaptive]</td>
<td>0.62</td>
<td>94456</td>
<td>2069</td>
</tr>
<tr>
<td>CMAES</td>
<td>[(51), relu, adam, 0.0001, 28, constant]</td>
<td>0.62</td>
<td>7810</td>
<td>330</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>[(74, 22), tanh, sgd, 0.00060, 67, adaptive]</td>
<td>0.61</td>
<td>6583</td>
<td>206</td>
</tr>
</tbody>
</table>

5.2 Multi-layer Perceptron Regressor (MLPR)

Multilayer Perceptron is a supervised learning algorithm that fits a non-linear function between the features and labels [20]. The hyper-parameters that we are optimizing are the number of Layers, number of neurons in each layer, activation of each layer, solver, alpha, batch size, and learning rate. We used scikit – learn score function to evaluate the model. This function returns the mean accuracy on the given test data and labels. You can find the results in table 5.1.

5.3 Linear Regression

Linear regression [1] fits a linear model between the features and labels. Since this model in scikit – learn [25] only has two Boolean hyper-parameters, fit_intercept (enable and disables calculating intercept for this model) and normalize (if set to true it will normalize the features before applying the regression). We use this model’s performance as a baseline to compare other models to it. Since there is only 4 different options using Grid Search is the best way to find the optimal hyper-parameters. You can find the result in table 5.2.
Table 5.2: Linear Regression on the Abalone Dataset.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters (fit_intercept, normalize)</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>(False, True)</td>
<td>0.57</td>
<td>$10^{-3}$</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.3: HPO for Stochastic Gradient Descent Regression on the Abalone dataset. Hyper-parameters are (loss, penalty, alpha, l1 ratio, fit intercept, epsilon, learning_rate). Default parameters are (squared_loss, l2, 0.5, True, 0.1, invscaling) with score = 0.517

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>[squared epsilon insensitive, l1, 0.0001, 0.3, False, 0.1, adaptive]</td>
<td>0.562</td>
<td>204</td>
<td>960</td>
</tr>
<tr>
<td>CMAES</td>
<td>[squared epsilon insensitive, elasticnet, 1.007, 0.54, False, 0.0012, invscaling]</td>
<td>0.538</td>
<td>1924</td>
<td>4545</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>[epsilon insensitive, elasticnet, 1e-05, 1, False, 1.0, constant]</td>
<td>0.559</td>
<td>174</td>
<td>206</td>
</tr>
</tbody>
</table>

5.4 Stochastic Gradient Descent Regression

Stochastic Gradient Descent Regression is another linear model that minimizes a regularized empirical loss with Stochastic Gradient Descent. For this model we optimized, loss function, penalty, alpha (constant that multiplies the regularization term), l1 ratio (The ElasticNet mixing parameter, fit intercept (Whether the intercept should be estimated or not), epsilon (determines the threshold at which it becomes less important to get the prediction exactly right), and learning rate. You can find the optimization results in table 5.3.
Table 5.4: HPO for Elastic Net on the Abalone dataset. Hyperparameters are (alpha, l1 ratio, fit intercept, normalize). With default parameters (1, 0.5, True, False) with score = 0.336

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>[0.5, 0.5, False, True]</td>
<td>0.398</td>
<td>45</td>
<td>200</td>
</tr>
<tr>
<td>CMAES</td>
<td>[0.46, 0.30, False, True]</td>
<td>0.404</td>
<td>655</td>
<td>4040</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>[0.1, 1.0, False, True]</td>
<td>0.524</td>
<td>137</td>
<td>206</td>
</tr>
</tbody>
</table>

5.5 Elastic Net

Elastic Net is another linear regression model with combined L1 and L2 priors as regularizers. For this algorithm we optimized, alpha (Constant that multiplies the penalty terms), l1 ratio (The ElasticNet mixing parameter), fit intercept (Whether the intercept should be estimated or not), and normalize (if set to True, it normalizes the features). You can find the optimization results in table 5.4.

5.6 Bayesian Ridge Regression

Bayesian Ridge regression fits a Bayesian ridge model to our data. For this algorithm we optimized, $\alpha_1$ (shape parameter for the Gamma distribution prior over the alpha parameter), $\alpha_2$ (inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter.), $\lambda_1$ (shape parameter for the Gamma distribution prior over the lambda parameter), $\lambda_2$ (inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter.), fit intercept (Whether the intercept should be estimated or not), and normalize (if set to True, it normalizes the features). You can find the optimization results in table 5.5.
Table 5.5: HPO for Bayesian Ridge Regression on the Abalone dataset. Hyperparameters are \((\alpha_1, \alpha_2, \lambda_1, \lambda_2, \text{fit intercept, normalize})\). With default parameters \((1e-6, 1e-6, 1e-6, 1e-6 \text{ True, False})\) with score = 0.522

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>([1e-7, 1e-6, 1e-6, 1e-7, \text{False, False}])</td>
<td>0.587</td>
<td>91782</td>
<td>35496</td>
</tr>
<tr>
<td>CMAES</td>
<td>([1e-6, 9e-7, 8e-7, 6e-7, \text{False, True}])</td>
<td>0.566</td>
<td>865</td>
<td>5220</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>([5e-5, 5e-5, 6e-5, 0.0002, \text{False, False}])</td>
<td>0.571</td>
<td>204</td>
<td>206</td>
</tr>
</tbody>
</table>

5.7 Gradient Boosting Regression

This algorithm optimizes an ensemble of weak learners [10]. For this algorithm we optimized, loss function, learning rate, number of boosting stages, sub sample (The fraction of samples to be used for fitting the individual base learners), criterion (The function to measure the quality of a split), and maximum depth (Maximum depth of the individual regression estimators). You can find the optimization results in table 5.7.

5.8 Epsilon-Support Vector Regression \((\epsilon\text{-SVM})\)

\(\epsilon\text{-SVM}\) fits a linear or non-linear function to the the data, constructs an \(\epsilon\)-insensitive tube, and only penalizes the objects that lie outside of this tube [18]. For this algorithm we optimized, kernel, degree of polynomial kernel, C value (Regularization parameter), gamma (Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’ kernels), coef0 (Independent term in kernel function), and the epsilon in the epsilon-SVR model. You can find the optimization results in table 5.7.
Table 5.6: HPO for Gradient Boosting Regression on the Abalone dataset. Hyperparameters are (loss function, learning rate, number of boosting stages, sub sample, criterion, max depth). With default parameters (ls, 0.1, 100, 1, friedman_mse, 3) with score = 0.56

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>[ls, 0.1, 90, 1, friedman_mse, 4]</td>
<td>0.58</td>
<td>33949</td>
<td>961</td>
</tr>
<tr>
<td>CMAES</td>
<td>[ls, 0.2, 80, 0.8, mse, 3]</td>
<td>0.58</td>
<td>33756</td>
<td>186</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>[ls, 0.97, 251, 0.7, mse, 1]</td>
<td>0.57</td>
<td>25526</td>
<td>206</td>
</tr>
</tbody>
</table>

Table 5.7: HPO for Epsilon-Support Vector Regression on the Abalone dataset. Hyperparameters are (kernel, degree, C, gamma, coef0, epsilon). With default parameters (rbf, 3, 1.0, scale, 0, 0.1) with score = 0.54

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Optimal parameters</th>
<th>Optimal score</th>
<th>Total Time (s)</th>
<th>Total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridSearch (BFS)</td>
<td>[rbf, 2, 5.0, 0, 0, 0.5]</td>
<td>0.61</td>
<td>15000</td>
<td>35000</td>
</tr>
<tr>
<td>CMAES</td>
<td>[rbf, 2, 9.9, auto, 1.0, 0.2]</td>
<td>0.60</td>
<td>20089</td>
<td>5805</td>
</tr>
<tr>
<td>Bayesian Optimization</td>
<td>[poly, 3, 3.0, scale, 1.0, 0.0]</td>
<td>0.58</td>
<td>950</td>
<td>206</td>
</tr>
</tbody>
</table>
5.9 Chapter Conclusion

Based on our experiments, Multi-layer Perceptron Regressor 5.2 is the best performing algorithm for predicting abalone age with $score = 0.62$. Additionally, these experiments give us a better understanding of the performance of different optimization algorithms in different scenarios. As you can see from the tables above, the grid search finds the highest optimal score (except in one case 5.4w here the step size were not small enough to cover the optimal parameter values) as expected. However, in most cases, it uses much more computational power than the other two algorithms. CMA-ES also has a comparable performance with grid search, in the sense that the optimal score is equal or very close to grid search, however, in some cases, this algorithm also takes a long time to converge. Additionally, CMA-ES requires the user to specify a sigma, where sigma can be interpreted as the step size between each iteration. This additional parameter could be another hassle in optimizing the optimization itself which could be a drawback for CMA-ES. However, Bayesian optimization always (if the default setting is not optimal already) finds better hyper-parameter values than the default in a limited number of steps that can be specified by the user as well. For this experiment, we set the number of steps to 206. In conclusion, Bayesian optimization is a better choice as it doesn’t require the user to specify any parameters related to the algorithm and in most cases improves the performance from the default value.
Chapter 6

Conclusion

In this thesis we created a hyperparameter optimization framework that allows us and others to optimize hyperparameters of any opaque model (2). We then provided a thorough comparison and a detailed explanation of different optimization algorithms that have been widely used for hyperparameter optimization; Bayesian Optimization, Covariance Matrix Adaptation Evolutionary Strategy, and Grid search optimization to name a few of such algorithms (3).

Since this thesis was motivated by optimizing hyperparameters of NVIDIA’s localization algorithm, we tested our framework and the optimization algorithms on it. By doing so we realized that both CMA-ES and Bayesian optimization can improve the performance of the algorithm at least $10$ times faster than grid search. Additionally, we studied the search area of these algorithms, Bayesian optimization, and CMA-ES, and confirmed that they indeed focus their search in the right area where the optimal parameters lie with high probability (4.2). Additionally, we realized that the main factor in increasing optimization time is the evaluation of the opaque model on the full dataset. Thus once we confirmed that our framework works and the optimization algorithms improve the performance, we decided to also tackle another change and lower the optimization time even further. Our idea was to find a sub-set of our data that we can use to evaluate the performance of our opaque model given the hyper-parameter values. The data we were working with were videos of the car driving in different areas; hence our assumption was that if we divide the videos into segments
and pick the segments that represent the full data set we might be able to optimize the opaque model but evaluate it on a smaller data set and as a result lower the evaluation time. In 4.3 section we explained some of the methods we employed and their result. After exploring different options, we realized that while our sub-data selection methods lower the computation time, the optimal hyperparameter values found on the sub-data don’t always, or with high confidence, perform better than the default when evaluated on the full data. Therefore, we decided not to include the sub-data selection into the optimization pipeline for NVIDIA’s localization algorithm.

In addition to testing our framework on NVIDIA’s algorithm, we evaluated the performance of the optimization algorithms for predicting abalone age [7]. From these experiments, we concluded that Bayesian optimization is the best choice among the other available optimization methods. Bayesian optimization achieved at least 2% and even 20% improvement for Elastic Net regression 5.4). These results are helpful to many researchers who aim to optimize the hyperparameter values in their models.

6.1 Future Work

While our work was promising in showing the effect of HPO on different models and areas, there are still chances for improvement. For instance, the sub-data selection methods that we explored in section 4.3 can be extended to other problems and models in which their evaluation is also costly. For instance, in chapter 5 we used abalone dataset [7] which is a relatively small dataset (4177 samples), compared to some computer vision datasets such as COCO 2017 [27] that has around 121000 samples. While training abalone is in the order of minutes, training models such as ImageNet for detecting objects can take in the order of days [29]. As a result, having a robust sub-data selection that can reduce the opaque model evaluation time can be very useful in a broad range of applications.
Appendix A

Appendix

A.1 Result of HPO on Debiasing Clinical Trials

In another research we utilized our HPO framework to optimize the hyperparameter values of our debiasing model. The parameters that we optimized were: loss weights: \((w_1, w_2, w_3, w_4)\), number of latent dimensions, number of neurons in the first layer, and number of neurons in the second layer. With our initial best guess for each parameter our model had an \(\text{AUC} = 0.84\) and after optimizing with Bayesian optimization we could improve it to \(\text{AUC} = 0.86\).
Bibliography


