Università degli Studi di Bari Aldo Moro
Dipartimento di Informatica

Corso di Laurea in

Informatica e Tecnologie per la Produzione del Software

Insert Your Title Here

Filippo Chinni Carella

Supervisor:
Prof. Corrado Mencar

June 13, 2024

Acknowledgement

Do not forget to acknowledge the supervisor

Abstract

An *abstract* is a summary of a research article, thesis, review, conference proceeding, or any in-depth analysis of a particular subject or discipline. It is often used to help the reader quickly ascertain the article's purpose. Therefore, an abstract should be self-contained and understandable on its own. When used, an abstract always appears at the beginning of a manuscript, acting as the entry point for any academic paper or patent application. Abstracting and indexing services for various academic disciplines are aimed at compiling a body of literature for that particular subject.

Contents

Chapter 1

Introduction

In Machine Learning, the process of training a model involves the optimization of a set of parameters, called Hyperparameters, which are not learned during the training process. The optimization of these hyperparameters is a crucial step in the development of a machine learning model, as it can significantly affect the performance of the model. The process of finding the best hyperparameters for a model is called Hyperparameter Optimization (HPO).

At the current state of research, HPO is a flourishing field, with many different techniques available, ranging from simple traditional manual tuning to more advanced automated optimization algorithms.

1.1 Aims and Objectives

The aim of this thesis is to explore the field of HPO, by testing and comparing different optimization algorithms. The objectives of the thesis are:

- To establish the importance of HPO in Machine Learning and its impact on the training process of a model;
- To evaluate the effectiveness of different HPO algorithms, using an existing library for HPO;
- To adapt an existing optimization algorithm to make it suitable for HPO, and test its effectiveness; in particular, the algorithm to be adapted is the Particle Swarm Optimization (PSO);
- To integrate the adapted algorithm into the existing library for HPO, and test it on different datasets and models;
- To test the effectiveness of the HPO algorithms on a more complex dataset, testing the limits and advantages of HPO in a more challenging context. Specifically on a real world problem, in the field of Drone Vision for agriculture.

In order to achieve these objectives, a series of experiments will be conducted, where different techniques will be chosen (or developed), analysed and compared. The experiments will be performed mainly on simpler datasets and models to make the process more manageable and to focus attention on the optimization algorithms themselves rather than on specific problems.

Subsequently, with regard to the Drone Vision problem, the experiment will be conducted on a more complex dataset and model. The reason Drone Vision is sensitive to the optimization of hyperparameters is that the functioning of a Machine Learning model, especially Neural Networks which are the most used models in Drone Vision, is highly energetically and computationally expensive. Usually, drones have limited computational resources, therefore, they can hardly afford to run a complex model to perform tasks such as Semantic Segmentation. The proposed methodology aims to perform a weighted Hyperparameter Optimization, where the complexity of the model is penalized, so as to find the best trade-off between performance and computational cost.

One optimization technique that will get particular attention is the Particle Swarm Optimization (PSO), which is a population-based optimization algorithm inspired by the social behavior of flocks of birds. The algorithm will be adapted to make it suitable for HPO; and then tested, evaluated, and compared. PSO is currently one of the most popular optimization algorithms, therefore, it is interesting to observe its performance in the context of HPO, given its efficacy in searching optimal solutions in high-dimensional spaces.

1.2 Overview of the Thesis

The rest of the thesis is structured as follows:

- **Chapter 2** provides an overview of the background knowledge about the topic of Hyperparameter Optimization, presenting the state of the art.
- **Chapter 3** describes the methodology used in the experiments, including the models and the algorithms.
- **Chapter 4** presents the results of the experiments, and discusses them.
- **Chapter 5** concludes the thesis, summarizing the results and suggesting future work.

Chapter 2

Background

In Machine Learning, Hyperparameter Optimization or "Tuning" is the process of choosing the optimal Hyperparameters for the learning algorithm [1]. An "Hyperparameter" is a value used to control the learning process, basically a parameter of the Learning Algorithm and not of the model itself [2].

Hyperparameter Optimization has the goal to determine the best tuple of Hyperparameters that minimize the value of a loss function [3]. The process of selecting the right Hyperparameters configuration is called "Hyperparameter Optimization" or "Hyperparameter Tuning".

2.1 Hyperparameter Optimization

The Hyperparameters allow Machine Leaning models to be customized, or tuned, for specific tasks or datasets [2]. Basically, one important aspect of Hyperparameters is to allow a particular ML model's architecture to be reused in different situations and different datasets.

Generally, what each Hyperparameter does when considered alone is known; but it is hard to predict what the best combination of Hyperparameters is the best for a specific problem or specific dataset.

Hyperparameter Optimization is generated from the necessity to automate the tedious and expensive process of manual setting via trial-and-error the Hyperparameters of Machine Learning models [2]. Choosing the optimal architecture for the model is not an easy task, and in many cases the possibilities are so many that it is impossible for a human to select the best possible configuration. Thus, the ideal solution would be to make the machine to learn the optimal configuration by itself, exploring different model architectures to find the better performing one. Without automated Hyperparameter Optimization the programmer can only rely on trial-and-error and empirical experience.

Hyperparameter Optimization (HPO) algorithms have not only to automate the process of researching for the best Hyperparameters combination, but also to make this search computationally viable even for more complex models and problems. The main challenge of Hyperparameter Optimization is the arduousness to predict how the Hyperparameters could influence each other. This uncertainty makes the problem even more computationally complex, as in principle all possibilities must be explored.

2.1.1 Parameters and Hyperparameters

Machine Learning is about predicting and classifying data. ML models do so using parameters, meaning that the Models are Parametrized so that their behaviour is tuned for any given problem. The large term "Parameter" in Machine Learning can divide in two categories: Parameter (or Model Parameter) and Hyperparameter.

2.1.1.1 Parameters

A model Parameter is a configuration variable internal to the model. The Parameters are part of the model, are learned by training from data, they define the skill of the model and are thus required to make predictions. The Parameters in Machine Learning models have the same function that variables have in traditional programming.

Examples of Parameters in various Machine Learning models: Weights (in Neural Networks); Support Vectors (in SVM); Coefficients (in Linear or Logistic Regression).

2.1.1.2 Hyperparameters

A model Hyperparameter is a configuration that is external to the model. The Hyperparameters cannot be learned from the data and are often set by the programmer. Their values are either determined by heuristics or by optimization techniques.

Basically, Hyperparameters are not Parameters of the Model itself, but are rather Parameters of the leaning process/algorithm used to train the model.

Examples of Hyperparameters in various Machine Learning models: Learning Rate and Epochs (in Neural Networks); C and δ (in SVM); k (in k-nearest neighbors).

2.1.2 Basic Concepts of Hyperparameter Optimization

Two fundamental concepts of Hyperparameter Optimization, indispensable in the understanding of HPO, are the Objective Function and Search Space.

2.1.2.1 Objective Function

An Objective Function, in Optimization problems, is the function to optimize (the loss function is an example of Objective Function). More precisely, is a "scoring function",

it serves as a metric to define the performance of the trained model [3] [2].

The performance of a learning algorithm can be seen as a function: $f : \mathbf{X} \to \mathbb{R}$, that maps the Hyperparameters space $X \in \mathbf{X}$ to the result of the Validation function. Basically $f(X)$ represents an evaluation of the model trained with the tuple X of Hyperparameters.

The goal of Hyperparameter Optimization is to find the tuple X of Hyperparameters that permits to Minimize or Maximize (depends on the evaluation metric) the value of the Objective Function f.

2.1.2.2 Search Space

In order to search for the best combination of Hyperparameters, a Search Space has to be defined. The Search Space, or Configuration Space, denoted with $\mathbf{X}$, is the multi-dimensional set of all the possible values the Hyperparameters can be [2].

A Seach Space is a Multidimensional object, where each dimension represents a Hyperparameter, and each point represents a particular Hyperparameters configuration. More specifically, each point in the Search Space is a tuple, containing the values for each Hyperparameter of that combination.

2.1.3 Main Components of a HPO Algorithm

In general, all complex HPO algorithms two principal components: Samplers and Pruners (also called Searching and Scheduling).

The Sampler decides which configuration to try. The Pruner decides how to allocate the computational budget and when to stop a Trial, where a trial is the process of training a model with a particular configuration of Hyperparameters. In other words, is the evaluation of one single candidate solution.

On top of these two components, it is to be considered the Tuner, the object that executes the optimization process.

2.1.3.1 Sampler

A Sampler, or Searcher, or Search Algorithm, in HPO is the component of the HPO algorithm which selects ("samples") the next candidate configurations to try and evaluate. Simple Samplers like Grid Sampler (for Grid Search) and Random Sampler (for Random Search) just select candidates sequentially and randomly respectively; more complex Samplers, used for example in Bayesian Optimization, make the decision based on previous trials.

In the making of the choice of the Sampler, it is important to understand that Hyperparameters not always have the same importance in the results of the model. Also,

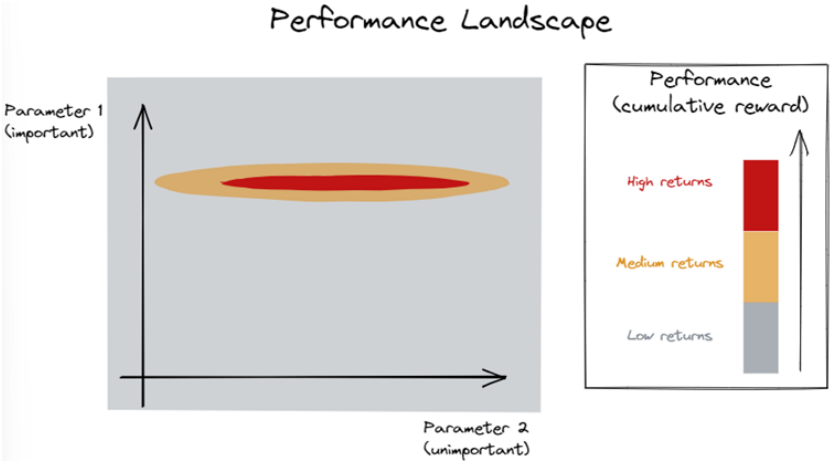

Figure 2.1.1: *A Graph showing how the importance of Hyperparameters can be spread in the Search Space. Source: [4]*

some Samplers are more effective with certain models than others. (See Fig. 2.1.1)

In HPO, the "importance" of an Hyperparameter over one other, is the relative power it has in determining the final value of the Objective Function; the more a Hyperparameter is important, the most impact a change of its value will influence the score.

2.1.3.2 Pruner

A Pruner, or Scheduler, in HPO is the component which determines if and when a trial needs to be stopped in order to dedicate more resources to more promising candidates. The Pruner has the goal to identify and then discard the worst performing configurations. The final objective is to ensure more computational resources to the remaining configurations.

It is important to understand when to prune. If pruning too early, the selection would be too premature, and some potentially good configuration may be discarded. If pruning too late, resource would have been wasted on bad configurations.

The most basic Pruners, on which more complex Pruners are based, are Median Pruner and Successive Halving. The Median Pruner is the simplest Pruner. At each pruning step, all the configurations with worse performance than the median of the results in that step, get discarded. (Fig. 2.1.2)

Successive Halving is a slightly more complex Pruning algorithm than Median Pruner, actually it can be interpreted as an generalization of the Median Pruner. The idea is simple: it starts with some configurations, at some point they get pruned, and only the most promising ones remain. The complex part is that this whole process is determined by Hyperparameters. Which are: the minimum budget, the initial number of trials,

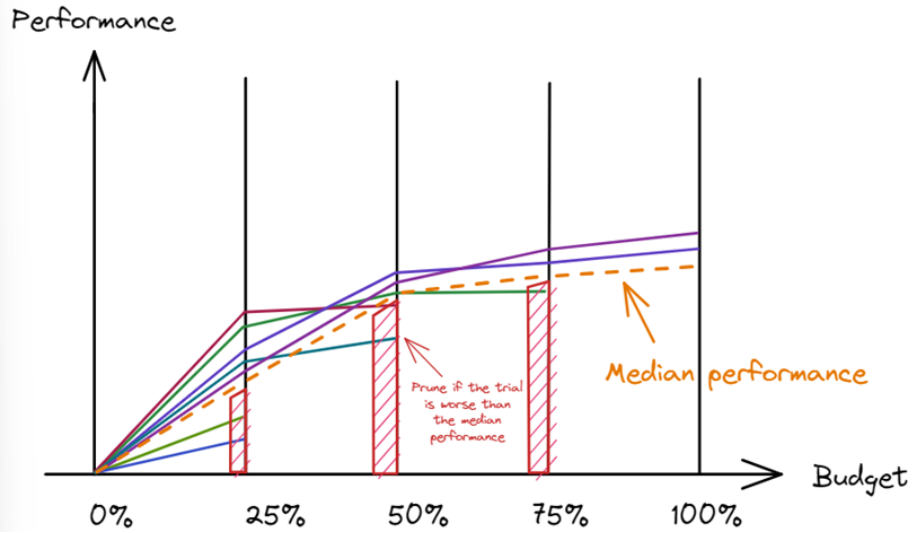

Figure 2.1.2: *A Graph showing the generic functioning of a Pruner Algorithm; in this case, a Median Pruner is showed. Source: [4]*

and the percentage of configurations to discard at each step.
More of Successive Halving Algorithm and Pruning algorithms in general will be discussed in the following sections.

2.1.3.3 Tuner

The Tuner, in HPO is the function that wraps up the Optimization Process. The Tuner runs the Sampler and the Pruner, manages the optimization of the process (like parallelizing), records the results and the possible errors of each trial. The Tuner can keep records of various information of the Optimization Process, such as errors, runtime, index of the configuration, suggestions of the Samplers, pruning actions of the Pruner.

2.2 Hyperparameter Optimization Algorithms

The goal of a Hyperparameter Optimization Strategy is to find the configuration of Hyperparameters, which leads, after the training of the model, to the minimum error and the maximum performance. There are many Hyperparameter Optimization techniques, some are specialized for specific ML models, other are more generic.

The HPO algorithms can be divided into two main categories: Black-Box Optimization algorithms, and Multi-Fidelity HPO algorithms (Fig. 2.2.1). The first category is focalized on the sampling mechanism, the second category is focused on Early Stopping and on the pruning mechanism.

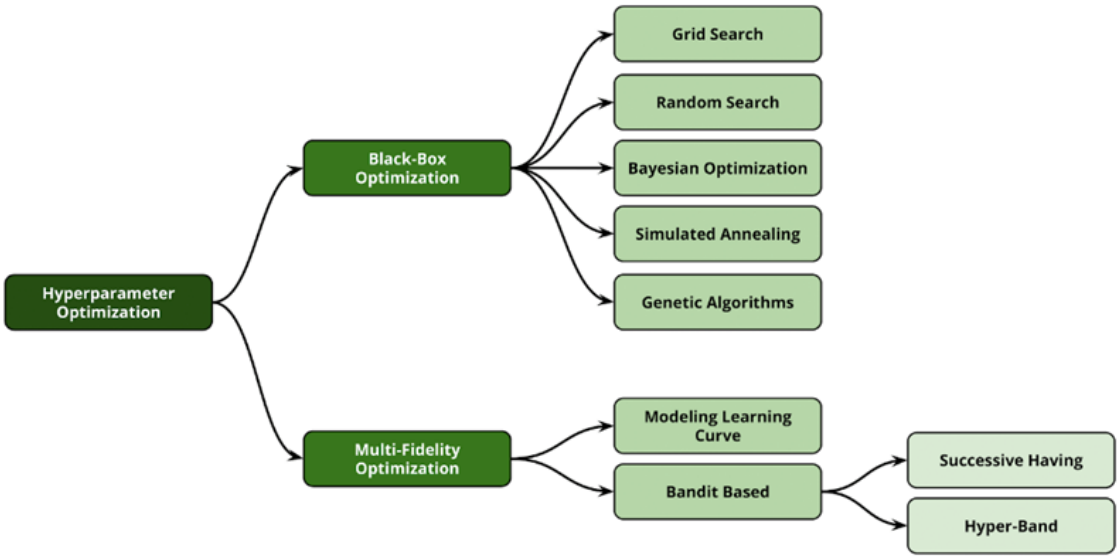

Figure 2.2.1: *Categories of Hyperparameter Optimization Algorithms. Source: [5]*

2.2.1 Grid Search

Grid Search is a traditional way to perform Hyperparameter Optimization. It simply consists of an "Exhaustive Search" of the Optimal Hyperparameters within a subset of specified Hyperparameters.

Grid Search is an approach that methodically builds and evaluates the model for each possible combination of Hyperparameters' values (Fig. 2.2.2).

The performance of a determined configuration of the Hyperparameters for the training of the model is evaluated using techniques such as Cross-Validation or Hold-One-Out. This evaluation is done for each tuple of Hyperparameters. At the end, the configuration with the best performance is the optimal configuration of Hyperparameters for that model.

For each Hyperparameter, the programmer defines the range of possible values. In the Grid Search technique, the Cartesian Product is applied to obtain all the possible combinations of values [6]. The output is the configuration of Hyperparameter which achieved the highest score, the best performance.

The main advantage of Grid Search is that is an "Embarrassingly Parallel" problem, which means that it is very easy to parallelize the operations. The strongest limitation, on the other hand, is that this technique suffers from the "Curse of Dimensionality", as its complexity tends to scale up exponentially very easily, making search intractable even for mid-sized HPO problems [1].

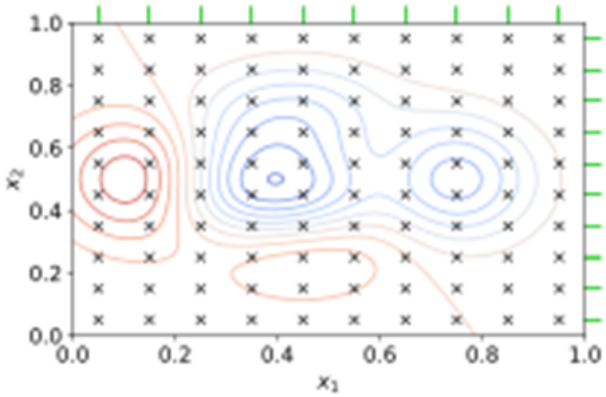

Figure 2.2.2: *Distribution of Candidate Configurations over the Search Space in Grid Search. Source: [3]*

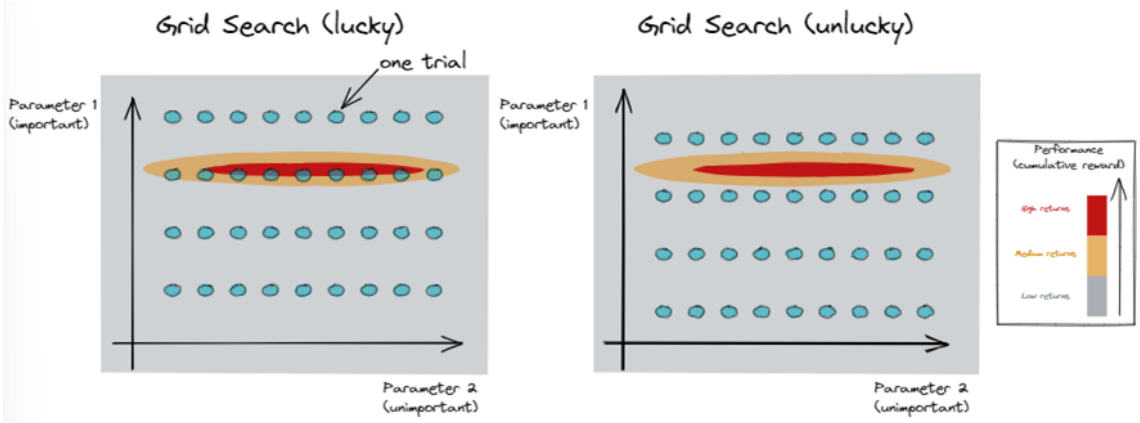

Figure 2.2.3: *Graph comparison showing the phenomenon of "Unlucky Seach" in Grid Search. Source: [4]*

Also Grid Search can suffer from "Unlucky Search", where none of the possible combinations obtained from the Param Grid obtains high results, because the particular candidate solutions obtained from the Cartesian Product do not cover the areas of the space where the performance is at maximum. (Fig. 2.2.3)

2.2.2 Random Search

Random Search is an alternative Hyperparameter Optimization technique to Grid Search [7]. It replaces the "Exhaustive Enumeration" of all possible Hyperparameter combinations with a simple Random Selection of these combinations (Fig. 2.2.4).

Random Search is an approach where rather than defining determined values for the Hyperparameters, statistical distributions are defined, which are used to randomly select values from them.

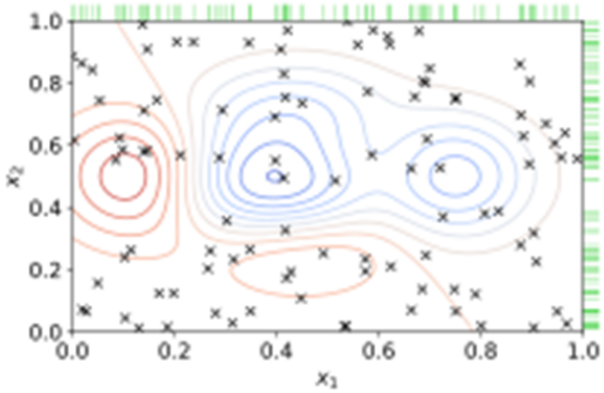

Figure 2.2.4: *Distribution of Candidate Configurations over the Search Space in Random Search. Source: [3]*

Differently from Grid Search, Random Search is an "Explorative" technique. It is able to explore much more values for all types of hyperparameters; this is due to the fact that the chosen points in the Search Space are not fixed in a Grid, which leaves out many possible values, but are randomly selected. It is considered an important baseline for comparing the performance of the most recent Hyperparameters Optimization techniques.

Random Search can outperform Grid Search, especially when the number of Hyperparameters which heavily effects the learning process is small [7]. Like Grid Search, Random Search is also an Embarrassingly Parallel problem.

Despite these advantages, Random Search is still very basic, it does not guarantee to converge to the best solution. It is not rigorous, meaning that there is no strategy to converge to the best possible solution, and no strategy to learn from the previous trials, adapting to the problem.

Random Search can resolve the "Unlucky Search" problem of Grid Search (Fig. 2.2.5). It also is suffering less than Grid Search of the "Curse of Dimensionality", that is, when there are many dimensions involved.

The advantage of Random Search lies in the fact that Hyperparameters, most of the time, are not equally important, and so strategies that examine all the possibilities, are most of the times losing time on useless evaluations [6].

Grid Search and Random Search are the simplest Hyperparameters Optimization techniques, so they are utilized when the problem is fairly simple and does not involve high complexity. Also, both Grid and Random Search tend to become very expensive when the complexity grows, Grid Search especially [8].

In general, Grid Search is more appropriate for problems where the best values range for a Hyperparameter is already known, and so the search is about finding the best

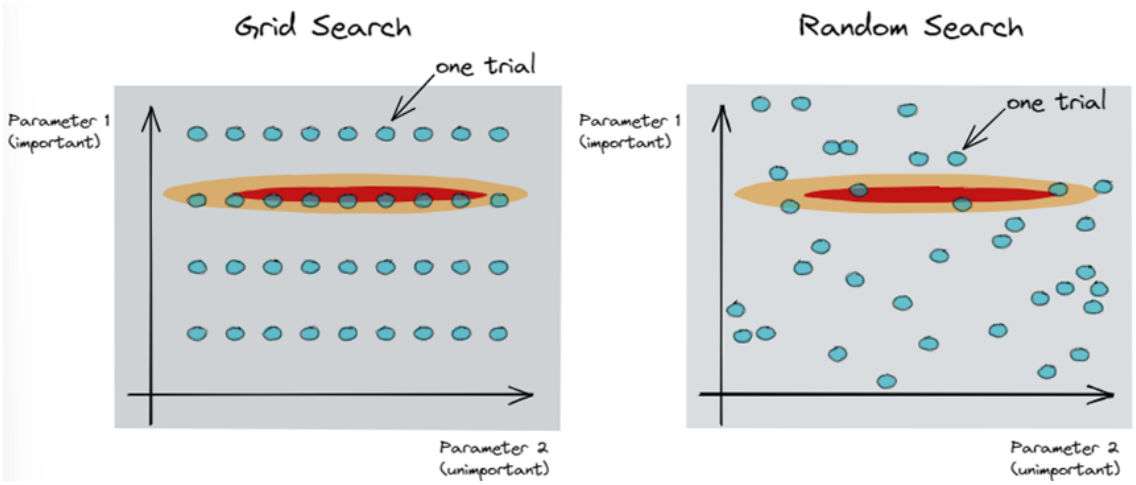

Figure 2.2.5: *Graph comparison between the distribution of candidate configurations in Grid Search and Random Search. Source: [4]*

combination between those already good-performing values.

Random Search on the other hand, is more appropriate for discovering new Hyperparameters values or new unexpectedly good combinations.

2.2.3 Bayesian Optimization

Bayesian Optimization is a Hyperparameter Optimization technique which builds a probabilistic model of the target function mapping the Hyperparameters values to the performance obtained on the Validation Set [9].

The former simpler techniques, like Grid Search and Random Search, do not learn from the previous trials, wasting useful information about the Search Space. The following techniques, starting from Bayesian Optimization, adapts to the problem, learning from the previous trials.

Basically, this method smartly explores the potential optimal configurations of the Hyperparameters' values, basing its decision on the performance of the previous configuration. (Fig. 2.2.6)

In HPO, an important trade-off is the one between Exploitation vs Exploration. Exploitation takes advantage of those values which are already expected to be close to the optimum. Exploration is the concept of trying new unexplored risky values which could improve the performance and add new knowledge about a particular Hyperparameter.

Bayesian Optimization balances Exploitation and Exploration, considering both concepts in order to get closer to the Optimum.

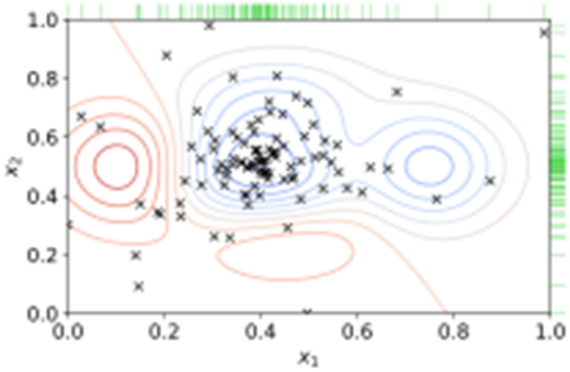

Figure 2.2.6: *Distribution of Candidate Configurations over the Search Space in Bayesian Optimization. Source: [3]*

The main idea of Bayesian Optimization is to estimate the performance of a certain configuration before that configuration gets tried. The Bayesian Optimization starts randomly from one configuration, or from a configuration which is already known to be at least decent. Then a probabilistic model (Acquisition Function) of the Objective Function is built, the idea is to predict which configuration will maximize the value of the performance, the found configuration will be the next. At each iteration the Acquisition Function is updated, and thus, the prediction becomes more and more accurate. Bayesian Optimization can find the best Hyperparameter Configuration much faster than traditional search algorithms.

2.2.4 Gradient-based Optimization

Gradient-Based Optimization is a Hyperparameter Optimization technique used with specific Machine Learning model such as Neural Networks and SVMs [10] [3].

The Hyperparameters are optimized just like the model Parameters, with Gradient Descent. The more traditional approach of this optimization algorithm uses "Automatic Differentiation" to calculate the so called "Hypergradients", to update the values of Hyperparameters. Basically, the Gradient of the model is also used during HPO.

While using this method would actually be more efficient, it requires to calculate second-order derivates, which are computationally expensive to approximate. These techniques are therefore very rarely used, because are difficult to apply.

More recent and complex techniques are now being developed for the Gradient-Based Optimization, making it more efficient and able to work with Discrete Hyperparameters [11].

2.2.5 Evolutionary Optimization

Evolutionary Optimization is a Hyperparameter Optimization technique which follows a process inspired by biological concept of Evolution. Evolutionary Optimization is applied in almost all fields of Machine Learning, from Statistical ML to Neural Networks and especially in Deep Learning [3] [12] [13].

Steps of Evolutionary Optimization:

Evolutionary Optimization consists in a series of steps, each of which is inspired by a step of the Theory of Evolution:

1. Create an initial population of random solutions (random Hyperparameters configurations)
2. Evaluate the Hyperparameters configurations and obtain their "Fitness Function" (a particular type of Objective Function that summarizes the results of each score metric)
3. Rank the Hyperparameters tuples by their Fitness Function value.
4. Replace the worst tuples with new tuples generated through crossover and mutation (where Crossover and Mutation are similar concepts as they are in Biology)
5. Go back to step 2 until the performance goal of the algorithm is reached or until the performance is no longer improving.

2.2.6 Population-based Training

Population-Based Training (PBT) is a Hyperparameter Optimization algorithms family; it is a broader optimization technique, applied to learn both Hyperparameters values and model's parameters.

Is is very similar to Evolutionary Optimization, actually the latter is a particular case of Population-Based Training.

PBT is very flexible and can be applied to a wide range of different models or problem domains. The optimization process does not depend on aspects like Network Architecture, particular Loss Function, etc.

Main Difference with Evolutionary Optimization:

Like Evolutionary Optimization, Population-Based Training (PBT) involves iteratively replacing poorly performing configuration with better performing modified.

The first difference between PBT and Evolutionary Optimization is that in PBT are not necessarily used Evolutionary Algorithms, but rather a wider range of techniques

are applied. Another main difference is the fact that configurations at each generation are not replaced, but rather updated.

In PBT, when possible, the concept of "Warm Starting" is often used, where models are initialized with hyperparameters and weights from already known better configurations to speed-up the training process [3].

2.3 Multi-Fidelity Hyperparameter Optimization

Hyperparameter Optimization is one of the most computationally expensive processes in the Training of a Machine Learning model. At the start of Hyperparameter Optimization, a number of configurations to try is chosen. The ideal solution of course is to try every single configuration, to be sure to find the best one. But when the model is too complex, the computational budget is limited, and tring every single configuration fully is not the most efficient strategy. A promising strategy is to use Multi-Fidelity Hyperparameter Optimization.

Multi-Fidelity Hyperparameter Optimization is a HPO techniques family, or rather, are techniques that are often used in combination with the other approaches in order to improve the efficiency of the optimization process. These techniques are mainly applied for models that have a large number of Hyperparameters and a large number of possible values for them.

The main idea is to early-stop a trial of a particular configuration if that trial is not promising. At the start of the search, each configuration is given the same (low) amount of computational budget. At one point, the least promising configurations up to that point are abandoned, and the remaining are given more budget.

This process gets iterated, until at the last iteration only a small set of configurations will have the whole budget to complete their evaluation. (Fig. 2.3.1)

2.3.1 Introduction to Multi-Fidelity HPO

When applying for Hyperparameter Optimization algorithms on Neural Networks, the workload is much bigger than any other Machine Learning model. Using parallelization techniques enables to reduce the time of the processes, but the workload itself still remains the same.

With Multi-Fidelity Hyperparameter Optimization, it is possible to reduce the total computational workload the machine has to go through during the Optimization Process. Multi-Fidelity HPO uses pruning. Pruners determine if and when an unpromising trial needs to be stopped early, pruned.

Process of Multi-Fidelity HPO:

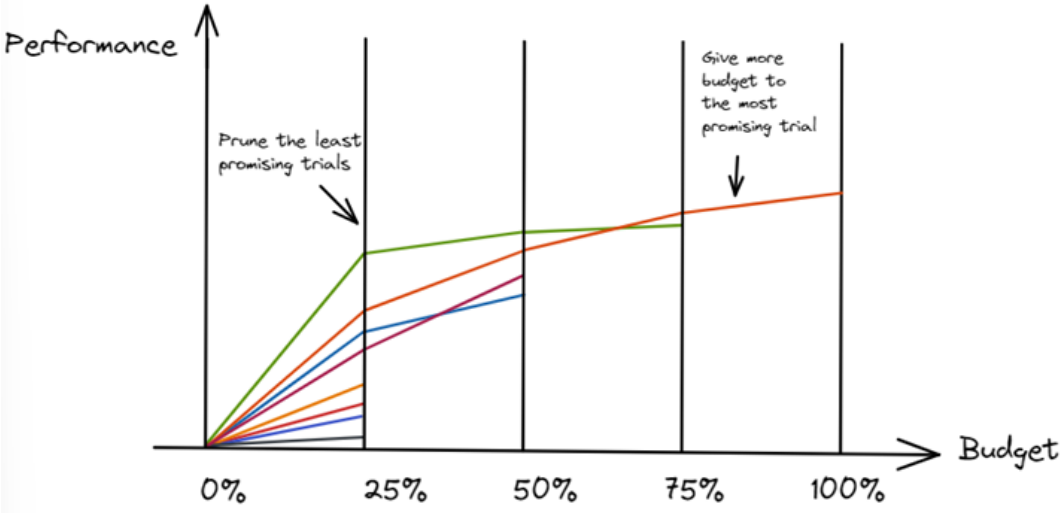

Figure 2.3.1: *Graph showing an example of an Multi-Fidelity HPO; this particular example could represent different types of Pruners as Median, SHA or Hyperband. Source: [4]*

After a few epochs of the trial of one Hyperparameters Configuration, it is already visually evident if that trial is or not promising. The Pruner algorithm decides when and how to stop those unpromising trials, allowing only the promising trials to complete the number of epochs and thus fully complete the training.

During each "Rung" ("Step") of the process, each trial is partially evaluated in terms of the Validation Error (or any other metric, evaluated on the Validation Set), which is used as an estimate of the final score of the configuration. The choice of which precise value the trial should reach to proceed to the next Rung, depends on the Pruning algorithm.

2.3.2 Multi-Fidelity HPO Algorithms

Multi-Fidelity HPO algorithms divides in Multi-Armed Bandit approaches, and Modeling Learning Curve approaches. We focus on the Multi-Armed Bandit (MAB) approaches. (Fig. 2.3.2)

The main characteristics of MAB algorithms are the Dynamic allocation of resources, and the balance between Exploration and Exploitation.

The most important MAB algorithms are: Successive Halving (SHA), Hyperband, Iterated Racing, ASHA, BOHB. The most basic Multi-Fidelity HPO MAB algorithm is Successive Halving (SHA). All the basic aspects of SHA, like the trade-off between trials and budget, are valid for all MAB algorithms.

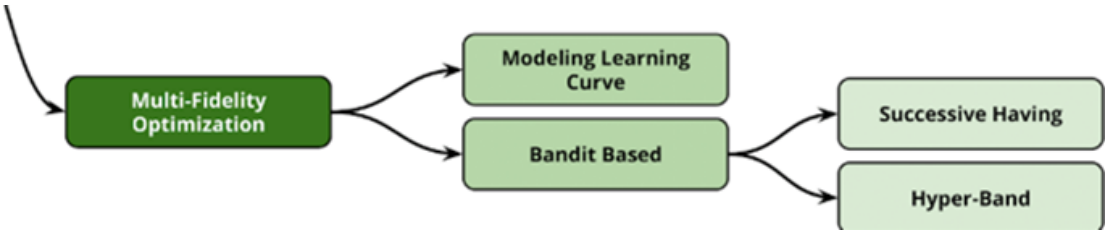

Figure 2.3.2: *Categories of Multi-Fidelity Hyperparameter Optimization Algorithms. Source: [5]*

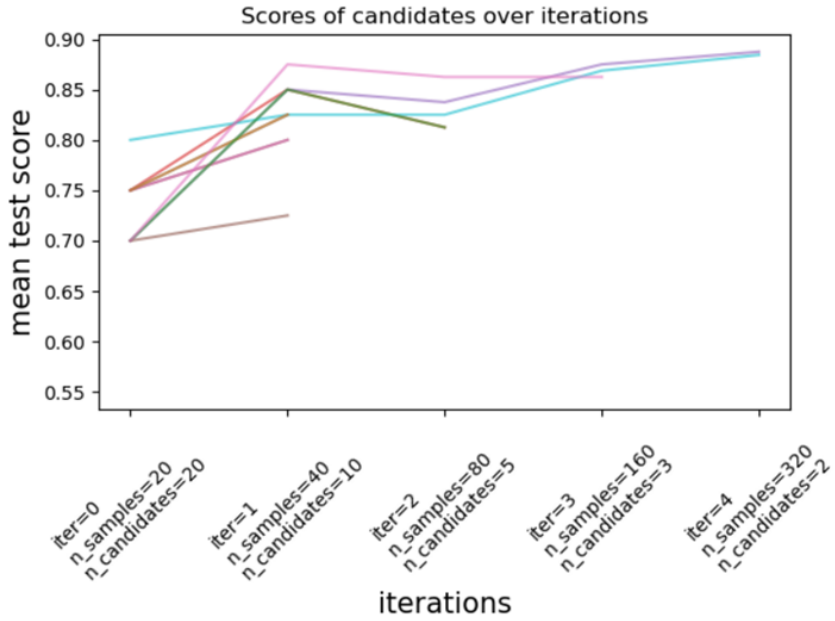

Figure 2.3.3: *Graph showing an example of an Optimization using Successive Halving Algorithm. The colored lines represent Trials. Source: [14]*

2.3.2.1 Successive Halving

Successive Halving can be seen as a tournament among the candidates (Hyperparameter Configurations) of the HPO process. The Successive Halving algorithm divides the "searching" process in iterations, that are often called "Rungs". At the end of each Rung level, only the best (most promising) candidates pass to the next level [14] [15]. (Fig. 2.3.3)

The two most important aspects of Successive Halving are the Number of Candidates and the Amount of Resources dedicated at each iteration. Each Rung has a number of Candidates and an Amount of Resources; the rule which determines how and how much those quantities should vary at each Rung depends on the Pruning algorithm.

The Number of Candidates and the Amount of Resources also determine how many Rungs the algorithm will have, and when each pruning step happens.

In simpler Pruning algorithms like Successive Halving, the parameter which control how much the two quantities should vary is called Factor.

Factor controls the rate at which the resources grow and the number of candidates decreases. At each Rung, the number of resources (per candidate) is multiplied by Factor, and the number of candidates is divided by Factor [15].

The resource parameter is the particular data to use as "budget". The choice of this "Resource" depends on the Machine Learning model object of the HPO.

For a model like RandomForest, a good resource is "n_samples", which is the fraction of the Dataset on which to train the model.

For Neural Networks the most used data a resource is the Hyperparameter "epochs". Basically, during successive halving, only the most promising configuration have the opportunity to complete progressively more epochs [15].

The parameter min_resources is the amount of resources allocated at the iteration (per candidate). This is a very important number for the success of the algorithm, as if it is too small, potentially good candidates may have a bad start and get pruned prematurely; if it is too big, the algorithm will be inefficient, as most candidates, even the worst ones, will consume a lot of resources at the first iterations [15].

2.3.2.2 Hyperband

The idea of Hyperband is to efficiently allocate resources to the Optimization Process, combining Random Searching, Successive Halving, and Early Stopping techniques [16] [10].

Hyperband work is especially utilized in settings where the amount of available computational resources is limited.

Starting from the total set of Hyperparameters Configuration, Early Stopping and Successive Halving are applied during the training process in order to maintain only the best promising configurations. At each iteration the amount of resources assigned to each trial is enlarged, allocating thus more resources to best promising configurations.

Hyperband outperforms most traditional HPO algorithms, it is usable for many ML models, it is scalable and parallelizable; it is particularly used in Deep Learning. Most modern studies are trying to combine Hyperband with Bayesian Optimization (BOHB), to obtain an even more powerful HPO algorithm.

2.3.2.3 Iterated Racing

Iterated Racing is a complex HPO algorithm based on efficient computational resources allocation to promising configurations [10].

The idea of Iterated Racing is to iteratively race configurations against each other, in pair, and then allocate more computational resources to the winning ones, discarding the losing. The configurations basically participate in a sort of tournament, which ends when a single configuration is remaining or other termination criteria are met.

2.3.2.4 ASHA and BOHB

Asynchronous Successive Halving Algorithm (ASHA) is the Asynchronous version of SHA [17]. Trials are executed asynchronously, gaining in efficiency, but increasing the risk of undeservedly promoting bad configurations to the next Rung.

Bayesian Optimization and Hyperband (BOHB) is an extremely complex HPO algorithm, which mixes ASHA, Hyperband and Bayesian Optimization [1] [18] [5]. Basically, BOHB is a combination of all the best performing HPO algorithms. It is probably, at least results-wise, the best HPO algorithm at the state-of-the-art.

2.4 Hyperparameter Optimization In-Depth

Hyperparameter Optimization is an expensive procedure, both computationally and effort-wise, so good approaches need to be used to make it more feasible.

Techniques like Parallelization makes the HPO algorithm more efficient. Overfitting mitigation techniques prevents the HPs to overfit the validation set. Good practices of HPO coming from Practical Aspects, allow the ML programmer to reduce the time of Trial-and-Error during the set of the algorithm.

2.4.1 Parallelization in HPO

Even when using complex HPO algorithms, the process can take hours or days to finish depending on how big the dataset is, how complex the model to train is, and how many Hyperparameters and Hyperparameter's values are there.

The computing time could be massively reduced by distributing the trials across parallel resources. There can be distinguish three different types of scheduling: Sequential, Synchronous and Asynchronous [2].

For what concerns parallelizazion in standard HPO algorithms, the focus of the parallelization is on the Sampling process:

- Sequential HPO: There is no parallelization, each trial is executed only after the previous trial.
- Synchronous HPO: The new batch of configurations is sampled when the previous set finishes. This means that if a batch requires more time to finish, the next batch will have to wait for the resource to be released.

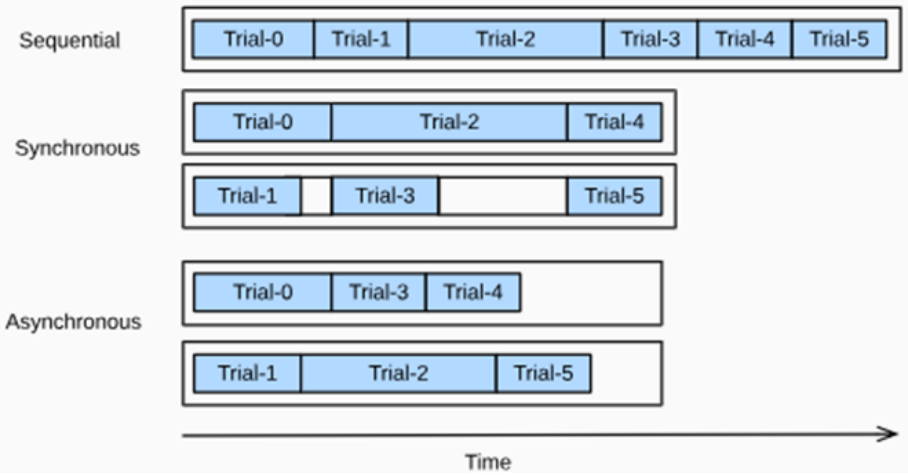

Figure 2.4.1: *Comparsion of Sequential, Synchronous and Asynchronous Optimizations over time. Source: [2]*

- Asynchronous HPO: Each new Hyperparameters configuration is evaluated immediately after the previous one is finished; so, each parallel worker is always busy [2]. (Fig. 2.4.1)

Pruning algorithms have to be parallelized just like Sampling algorithms:

- Synchronous Pruning Algorithms: a worker has to go through idle time in wait that all other workers on complete their subset of trials for the current Rung level. (Fig. 2.4.2)
- Asynchronous Pruning Algorithms: a subset of configurations is promoted as soon as a specified number of observations are collected at the current Rung level. This may indeed lead to reduce the accuracy of promotions, but the impact on the final result is modest [2]. (Fig. 2.4.3)

2.4.2 Hyperparameters Overfitting

When the Hyperparameter Optimization is completed, the chosen set of Hyperparameters are fitted on the training set. The risk is that the model is overfitting. The Hyperparameter Optimization may overfit the hyperparameters values to the validation set.

The solution is to evaluate the generalization performance (score) of the final model on a set that is completely independent to the one used to optimize the Hyperparameters. This can be done with Nested Cross-Validation, a more complex Cross-Validation technique.

In Nested Cross-Validation each fold in further divided into folds. The inner folds

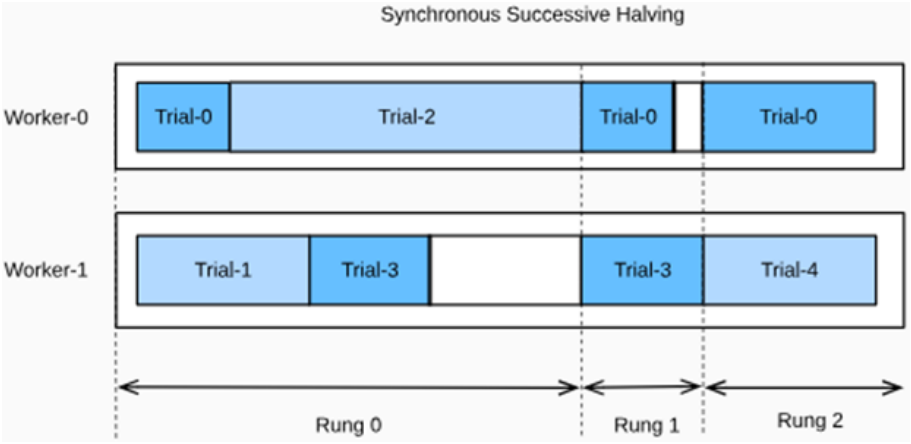

Figure 2.4.2: *Functioning of Synchronous Optimization in Multi-Fidelity HPO techniques. Source: [2]*

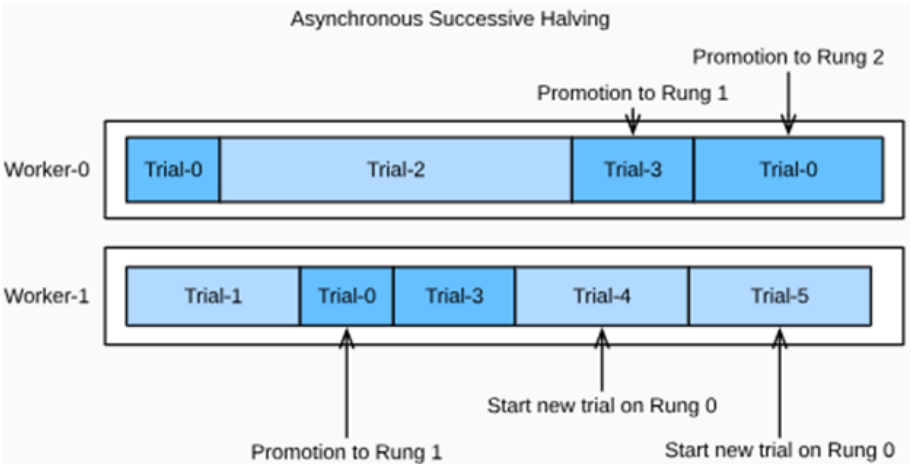

Figure 2.4.3: *Functioning of Asynchronous Optimization in Multi-Fidelity HPO techniques. Source: [2]*

are used to tune the Hyperparameters, while the outer folds are used to evaluate the model's performance as usual. (Fig. 2.4.4)

2.4.3 Practical Aspects of HPO

Choosing Evaluation Method:

One often forgotten component of a HPO algorithm is the Evaluation method of the model or trial. An ideal evaluator should be accurate, fast and simple. However, a trade-off remains between accuracy and speed; the more accurate, the probably more expensive it would be.

The most used evaluation method is to evaluate the current trial using the target

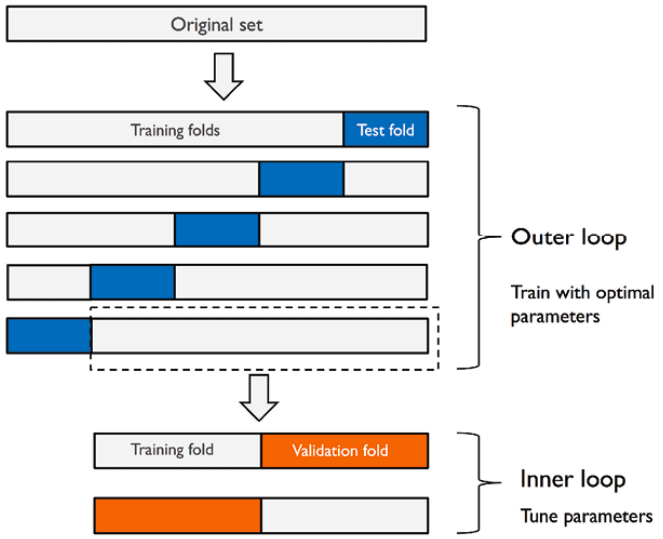

Figure 2.4.4: *Simple visual representation of Nested Cross-Validation. Source: Google Images*

Dataset. Although this is by far the most precise method, it can be slow and resource wasting. Especially in DNNs tuning, where the number of configurations is very high, it is too expensive to evaluate traditionally each trial.

Techniques like Early Stopping Strategies, which evaluate the model performance with an estimate function, rather than the actual evaluation function, allow to save time and resources during the evaluation process [10].

Choosing Performance Metrics:

The choice of the Performance Metric to use for an Optimization Process, should derive from the real world context the model will work on. In general, the most used metric is Accuracy, but in some application contexts it may be inappropriate, and depending on the situation, Precision or Recall could be better; also F1-Score if both Precision and Recall are needed.

Nevertheless, it remains true that for the most applications, it is impossible to determine a single metric able to capture all aspects of a model quality in a balanced manner [10].

Choosing the Search Space:

Hyperparameters can be classified based on their domain of possible values. Speaking of numeric HPs, there are mainly two different situations: Bounded in a closed interval (example: $[a, b]$); Bounded from below (example: $[0, \infty]$).

HPs bounded from below can be tuned without modifications. HPs bounded in a closed interval need to be tuned on a logarithmic scale (example: $[\log a, \log b]$), in order to optimize performances; this results in a shrinking, withouth losing effectiveness, of the interval.

The size of the Search Space is also important (Here with "Search Space" is not meant the theoretical complete Search Space of all possible configurations, but only the portion of that space considered for the problem in question). If the Search Space is too small, it may not contain well-performing Hyperparameter Configurations. If there are too many possible values for each HP, the cost of the search becomes too big. If there are too many HPs to tune, the algorithm will suffer from the Curse of Dimensionality. Because of the problems coming from the size of the Search Space, it would be better to tune as few HPs as possible, each having as few possible values as possible [10].

Choosing the HPO Algorithm:

The principal factor which determines which HPO algorithm to choose is the number of Hyperparameters.

With a very small number of Hyperparameters, between 2 and 3, Grid Search could be the best option as it searches all the possible combinations; furthermore, is the easiest algorithm to interpret and execute.

As the number of HPs raises, between 3 and 10, Bayesian Optimization is the best option.

If the number of HPs is very high, but the number of impactful HPs is low, then Random Search and Hyperband are the best choices.

For Search Spaces which are both very large and complex, Evolutionary Algorithms and Iterative Racing are the best choices.

One other factor that influences the algorithm choice is expensiveness of performance evaluation. If the evaluation process is expensive, then Multi-Fidelity algorithms like Hyperband are the best choices [10].

Choosing when to Terminate the HPO:

There are different options for this problem.

Option 1: defining an amount of runtime after which to stop the HPO, solely based on empirical data and intuition.

Option 2: setting a lower bound regarding the generalization error. The problem with this solution is that the specified value could never be reached or could take the algorithm too much time to reach.

Option 3: if no significant progress is made in a specified amount of time, the process is stopped. This option risks to stop the process too early.

Option 4: (only for Bayesian Optimization and similar) when the acquisition function estimates that further progress is little or unlikely [10].

2.5 Hyperparameter Optimization Libraries

Hyperparameter Optimization, as explained before, is an extremely computationally expensive operation, therefore each HPO algorithm needs to be the more optimized as possible. Furthermore, HPO is a boring operation for the ML programmer, who seeks more automated approaches than Trial-and-Error and rerunning.

For these reasons, libraries specifically designed to implement the HPO process exist. Wrapping all HPO components and approaches in a single aimed library, allows to make the process the most optimized and organized as possible.

2.5.1 Scikit-learn

Scikit-Learn is a Python API for Machine Learning. It provides different algorithms for Hyperparameter Optimization [19]. Scikit-Learn HPO algorithms versions are highly optimized and complete, including: parallelization, built-in resampling and OO approach.

GridSearchCV and RandomSearchCV:

The simplest algorithms are Grid Search and Random Search, Scikit-Learn offers the two respective classes `GridSearchCV` and `RandomSearchCV`. Both classes use Cross-Validation to evaluate the performance of the model, both classes require as input the model object of the tuning, and the Search Space, also called `param_grid`.

Other input parameters are: `cv` (number of folds for the Cross-Validation), `scoring` (the metric of evaluation used) and `n_jobs` (number of CPU cores on which to parallelize the search).

To start the Search for the best Hyperparameter combination, the `fit()` function must be called on the class object, passing as input X and y. The `fit()` method has as output the object result, which contains the tuple of the best Hyperparameters and the score that particular tuple achieved.

2.5.2 Optuna

Optuna is a Python API for Machine Learning. In particular, the main functions Optuna provides concern Hyperparameter Optimization [20].

Hyperparameter Optimization algorithms are often slow and poorly optimized. Moreover, these algorithms work better on some models and are bad on others. Optuna is specifically designed to work with any Machine Learning or Deep Learning Framework [21].

Optuna combines all the good practices for the quality and the efficiency of Hyperparameter Optimization. The search for the optimal Hyperparameters, for the best Hyperparameters combination is automated. Searching and Pruning are automated and use the most efficient algorithms. This allows to obtain the best efficiency. The parallelization process is easy and inexpensive, the search is run on multiple threads and multiple CPUs.

2.5.2.1 HPO in Optuna

Differently from most traditional approaches to Hyperparameter Tuning, Optuna divides the process in rigorous and ordered procedures.

The workflow for the use of Optuna, is divided into 3 phases:

1. Defining the Objective Function
2. Creating a Study Object
3. Running the Optimization Process

Phase 1 - Defining the Objective Function:

The first big difference compared to traditional Hyperparameter Optimization is that the Search Space is not created from examples but is based on Suggestions.

The Objective Function takes as an input a Trial object, which is Optuna's representation of the trial of a candidate solution; in the object are stored information like the tested hyperparameters, the fitness, statistics of the trial and other metadata. Is Optuna to manage the creation and the call of the Trial object, the user only has to define the Objective Function and define the ranges of values for the Hyperparamers on the Trial object.

On this object they can be called the `suggest_categorical()` and `suggest_float()` functions, respectively to generate suggestions (values) for Discrete and Continuous Hyperparameters. (There are also other suggestions like: `suggest_int()`, `suggest_loguniform()`, `suggest_discrete_uniform()`, `suggest_uniform()`. Each of which can take as input the low and high limits and the step).

Inside the Objective Function, there are not only defined the Hyperparameters, but there is also the code for the definition, training, and evaluation of the ML model object of the tuning.

Phase 2 - Creating a Study Object:

The study object is created via the `create_study()` function, which also defines the direction of the optimization (which depends on the evaluation criteria chosen).

For Classification's evaluation metrics, the direction has to be "maximize", for Regression's evaluation metrics, the direction has to be "minimize".

Phase 3 - Running the Optimization Process:

The Optimization Process runs start form the study object, using the method `optimize()`. The method takes in input the Objective Function and the number of Trials.

The method gives as output the number of the best trial, the accuracy of the best trial, and the tuple of Hyperparameters which represented the best trial.

2.5.2.2 Samplers in Optuna

In the field of study of Hyperparameter Optimization, Samplers define the way to sample the Hyperparameter values. Basically, the Samplers are the Search Algorithms. The `create_study()` function can take as input a Sampler, which will then be used as the search algorithm in the optimization process. Optuna offers different Samplers.

- **Tree-Structured Parzen Estimator (TPE) Sampler (TPESampler)**: It is the default Sampler that Optuna uses. It is a particular case of Bayesian Optimization technique. [9]
- **Random Sampler (RandomSampler)**: Samples Hyperparameters values randomly. It is equivalent to Random Search. [7]
- **Grid Sampler (GridSampler)**: Search across all different Hyperparameters in the Search Space. It is equivalent to Grid Search.
- **CMA-ES Based Algorithm (CmaEsSampler)**: Covariance Matrix Adaptation Evolution Strategy is a powerful search algorithm for continuous Search Spaces. [13]
- **Partial Fixed Sampler (PartialFixedSampler)**: Fixes the values of some Hyperparameters while optimizing the others. It is useful to explore the effects of single Hyperparameters or single smaller groups of Hyperparameters.
- **Nondominated Sorting Genetic Algorithm II Sampler (NSGAIISampler)**: Is utilized to optimize multiple objectives simultaneously. [12]
- **Quasi Monte Carlo Sampling Algorithm (QMCSampler)**: Is a less randomic version of the Random Search. Provides better coverage of the Search Space compared to Random Sampling. [7]

2.5.2.3 Pruners in Optuna

In the field of study of Hyperparameter Optimization, Pruners define the way to determine if and when to stop unpromising trials early. Pruners allow to reduce the amount of time and resources used for the optimization process. Are used especially in situations where the dataset is very large, or the model is very complex.

The textttcreate_study() function can take as input a Pruner, which will then be used to evaluate when to stop early a trial.

Optuna offers different Pruners: (Although SuccessiveHalvingPruner and HyperbandPruner outperforms all the other, and are thus the only actually useful ones)

- **MedianPruner**: is based on the Median Pruning algorithm.
- **NopPruner**: is based on the Non-Pruning algorithm.
- **PatientPruner**: is a Pruning algorithm with tolerance.
- **PercentilePruner**: is a Pruning algorithm where the specified percentage of trials pass.
- **SuccessiveHalvingPruner**: is based on the Asynchronous Successive Halving algorithm. [15] [17]
- **HyperbandPruner**: is based on the Hyperband algorithm. [16]
- **ThresholdPruner**: is based on the Threshold algorithm.

2.5.2.4 Other Aspects of Optuna

Optuna is a big and powerful library, therefore offers a lot of specific functionalities.

Callbacks:

Optuna allows to use Callback functions during the Optimization Process. The defined Callback functions define a behaviour, an action, to execute at the end of each trial. One common example of a callback function in this situation is a print function that prints some information regarding the current trial.

Saving and Resuming Optimization Sessions:

Optuna allows to save permanently the results of an optimization, to interrupt an optimization session to resume it later, and even to share an optimization session across different machines. All of this is done through the use of an SQLite Database, on which to store the data related to an optimization session.

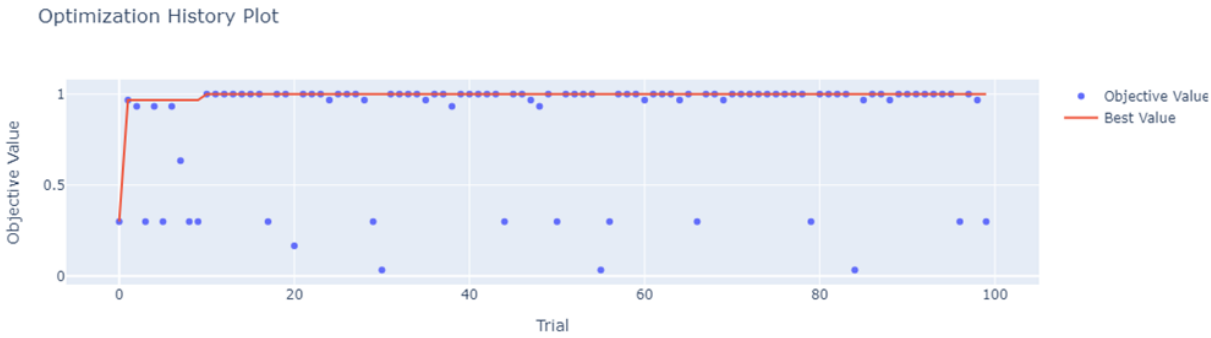

Figure 2.5.1: *Optuna Optimization History Plot*

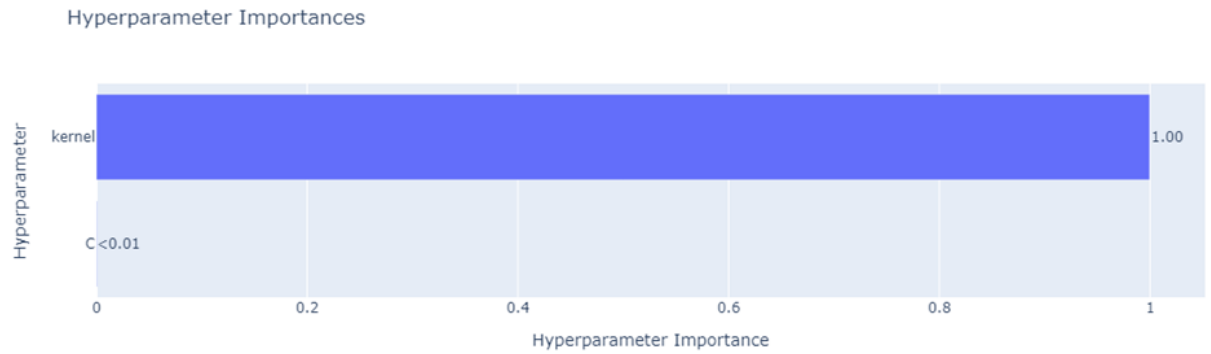

Figure 2.5.2: *Optuna Hyperparameters Importance Plot*

Feature Selection:

Optuna can be used in combination with apposite functions of Machine Learning libraries such as Scikit-learn to perform Feature Selection on the model in question. Using the Optuna's suggestion functions, is possible to also tune the function for the Feature Selection.

Visualizing Optimization Results:

Optuna preserves inside the study object some metadata that can be use to plot the history of the Optimization Process. Through the module Visualization, Optuna offers different functions to plot different aspects of the Optimization Process.

- History of the Optimization Process. (Fig. 2.5.1)
- Importance of some Hyperparameters over others. (Fig. 2.5.2)
- Relationship between Hyperparameters. (Fig. 2.5.3)
- Distribution of the trial result for each Hyperparameter value. (Fig. 2.5.4)

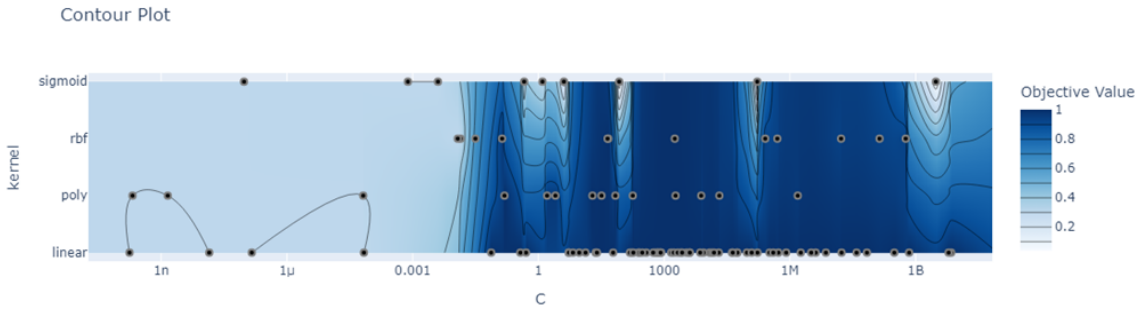

Figure 2.5.3: *Optuna Contour Plot*

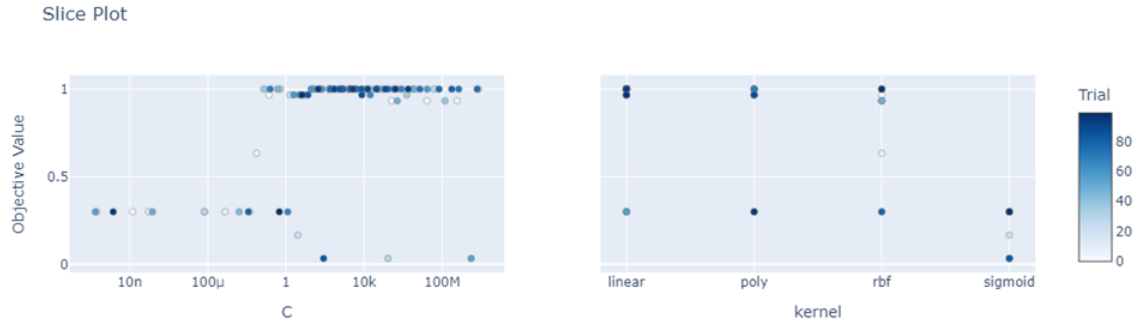

Figure 2.5.4: *Optuna Trial Distribution Plot*

2.6 Hyperparameter Optimization for Neural Networks

HPO is a very computationally expensive process, which cost progressively increments as the Search Space, and so the number of HPs considered enlarges [18]. Neural Networks, especially Deep Neural Networks, are the most complex type of ML model, and the number of HPs in a DNN model is very high.

Therefore, an important aspect of HPO is understanding which of the Hyperparameters have the stronger effects on Parameters (in the case of NNs, Weights) during the training.

2.6.1 Main Hyperparameters for Neural Networks

Hyperparameters, especially when talking about Neural Networks, can be divided into two categories: Hyperparameters for Training, and Hyperparameter for model design (Tab. 2.6.1). The most important Hyperparameters for both categories in Neural Networks are reported in the table.

Learning Rate:

Table 2.6.1: *Table showing the most common Hyperparameters in Neural Networks and their suggested ranges of values.*

Name	Type	Hyperparameter Range	log-scale
learning rate	float	$[10^{-6}, 10^{-1}]$	yes
batch size	integer	[8, 256]	yes
momentum	float	[0, 0.99]	no
activation function	categorical	{relu, sigmoid, tanh}	-
number of units	integer	[16, 1024]	yes
number of layers	integer	[1, 6]	no

Learning Rate is an Hyperparameter which determines the length of a "Learning Step", it basically represents the velocity of learning. The optimal value for the Learning Rate is different for every kind of problem, so in general only a few possibilities, one for each order of magnitude, are considered.

More complex algorithms, use techniques such as Learning Rate Decay, where the value of Learning Rate decreases over time during the training, additional Hyperparameters are also introduced to tune this sub-algorithm [18].

Optimizer:

Optimizer is the Hyperparameter which represents the algorithm used to update the Weights of the Neural Network. More specifically is that algorithm which is used to minimize the Loss Function during the training, calculating the gradients of the loss function.

The traditional Optimizer is Stochastic Gradient Descent; but nowadays there multiple more complex and better performing Optimizers, such as: Adam, RMSprop, Adagrad, Adadelta. Most complex Optimizers have their own Hyperparameters, which optimal values choice depends on the type of the model and its other Hyperparameters.

The choice of the Optimizer principally depends on the other HPs values. In general, Adam is well performing in most cases, RMSprop is good for Deep Networks [18].

Model Design Related Hyperparameters:

The two most important HPs of this category for Neural Networks are Number of Hidden Layers, Width of Layers (or Number of Neurons in each Layer).

The Number of Hidden Layers, generally speaking, grants the best accuracy results when it is a big number. Of course, this is a trade-off because the more Hidden Layers there are, the less lightweight the model will become, and so the more expensive it will

become to train and tune.

The Number of Neurons in each Layer, if it is too small will cause the model to underfit, if it is too big the model will overfit, and the cost of training would be too high. So, some suggestions can be followed [18]:

$$w_{input} < w < w_{output} \tag{2.6.1}$$

$$w = \frac{2}{3} w_{input} + w_{output} \tag{2.6.2}$$

$$w < 2w_{output} \tag{2.6.3}$$

Regularization:

Regularization is a Hyperparameters which represents the Regularization function, that is, a function used to mitigate the model's overfitting and reduce the Generalization error.

In general, the two most used Regularization techniques are L1 and L2, which both have their pros and cons, with L2 being the most used one. Although widely used for most ML models, for Neural Networks there are better alternatives.

Data Augmentation add fake synthetic data to the training Dataset, in order to enhance the generalization power of the model.

Dropout is the most used Regularization technique for NNs; during the training some Weights are "nulled", simplifying the model and making it less likely to Overfit the training set, improving the generalization power [18].

Activation Function:

Activation Function is a Hyperparameter which represents the mathematical function applied to the output of each Neuron. The goal of Activation Function is to introduce non-linearity into the network, without which the model could not learn complex patterns.

The most famous Activation Functions are the following: Sigmoid, Hyperbolic Tangent (tanh), ReLU, Softmax. Sigmoid works well for simple Neural Networks, is less efficient for more complex models. Softmax is only used in the output layer of NNs for Multiclass Classification.

The most used Activation Function nowadays is ReLU, especially convient for its simplicity. Many variants of ReLU have being developed in the recent years such as Leaky ReLU, PReLU, ElU, SeLU. Alternatives to ReLU are Maxout and Swish, which while they do resolve some problems of ReLU, are not that good for every situation [18].

	Advantage	Disadvantage	Applicability for DNN
Grid Search	- Simple - Parallelism	- curse of dimensionality	- Applicable if only a few HPs to tune
Random search	- Parallelism - Easy to combine with early stopping methods	- Low efficiency - Cannot promise an optimum	- Convenient for early stage
Bayesian optimization	- Reliable and promising - Foundation of many other algorithms	- Difficult for parallelism - Conceptually complex	- Default algorithm for tools - Variants of BO could be more applicable (TPE)
Multi-bandit methods	- Conceptually simple - Computationally efficient	- Balance between budget and number of trials	- Could be a default choice - Implemented by open-sourced libraries.
PBT methods	- Combine HPO and model training - Parallelism	- Constant changes to computation graph - Not extendable to advanced evolution	- For computationally expensive models

Figure 2.6.1: *Summary of main Hyperparameter Optimization Algorithms and their applicability to Neural Networks. Source: [18]*

2.6.2 HPO Algorithms applied to Neural Networks

A brief summary of the most important HPO algorithms, with advantages, disadvantages and applicability for Neural Networks, is showed in the following table: Fig. 2.6.1

Grid Search is applicable only when a just a small subset of the DNN's HPs is searched. Moreover, the user needs to already have knowledge about empirical good values of the selected HPs to further narrow the search.

Random Search can be used, only if combined with Early Stopping techniques, for the early stages of HPO.

Bayesian Optimization, at least in its original form, is not ideal for DNNs tuning. Variants like TPE (Tree-Structured Parzen Estimator), which introduce parallelization, are appliable to many DNNs models.

Multi-Armed Bandit algorithms (SHA, Hyperband, ASHA) are the default choice for DNNs' tuning. They outperform most algorithms in DNNs' models.

Population Based Techniques are a good choice for large models and large datasets [18].

Chapter 3

Methodology

This methodology chapter serves to introduce strategies, implementations, models and approaches that are put into practice in the Experiments chapter (Chapter 4). Therefore, it will include:

- **Particle Swarm Optimization:** an explanation of the most important of the sampling techniques utilized in the experiments.
- **Custom Implementation of PSO:** a detailed description of the design of the custom implementation of the algorithm, and some insights on the actual code implementation.
- **Enhancing Optuna Optimization:** a description of the design of the infrastructure built to run the experiments, in order to make Optuna more robust and reliable, and the experiments more comparable.
- **Applied Neural Network Models:** contains the list, with attached descriptions, of the machine learning objects involved in the experiments, along with descriptions of the training and evaluation of the utilized models.

Code Repository:

All the implementation details listed throughout this chapter refer to the following repository: [22]. The repository is divided into two main packages, `utils` and `experiments`.

The `experiments` package contains the code for the execution of the actual experiments and additional "private" code which is specific for the execution of the determined experiment.

The `utils` package contains code that is common to all the experiments, in order to make the project more maintainable and modularized.

All the code is written in Python (version 3.8 to 3.12). The required Libraries and Dependencies are listed in the `requirements.txt` file in the root of the repository.

3.1 Particle Swarm Optimization

3.1.1 Introduction to Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization approach originally proposed by Kennedy and Eberhart in 1995 [23]. The idea was to replicate the behaviour of certain animal species which move in groups, such as flocks of birds or shoals of fish.

The reason is the assumption that each singular individual in the group can take advantage from the collective experience of the entire group; basically, each individual can benefit from what it learns from other individuals, and in the same way, it can share its discoveries with the other individuals [24].

3.1.1.1 Particle Swarm Optimization as an Optimization Problem

Translating the previously explained behaviour into specific terms, each individual can be interpreted as an agent which "flies" in the Search Space, where each physical position is a candidate solution in the n-dimensional Search Space, and the best solution found by the entire group is the best solution.

The best solution found by the entire group is likely not the real global optimal solution. However, it is a solution that is near the global optimum.

The goal of Particle Swarm Optimization, as all optimization problems, is to find value which maximizes (or minimizes) the value of an objective (or Fitness) function f.

The Objective Function f is defined on a Search Space $\mathbf{X}$, the multi-dimensional vector representing all the possible solutions within the defined limits. The PSO algorithm will return the vector X, which represents a single solution, which maximizes (or minimizes) the value of f.

To find the maximum (or minimum) of the Fitness function f, the best solution would be to perform an exhaustive search on all the possible solutions. However, this approach is too computationally expensive, and basically inapplicable to higher-dimensional spaces [24].

Therefore, in PSO, the same way as a flock of birds searches for food, moving in the air, the algorithm starts with a number of random points in the search space, which are called Particles, and has them look for an optimal value by roaming in random directions.

At each atomic step, each Particle (individual) searches for its Local Optimal value, basing its research on both the current Local Optimum, and the current Global Optimum of the whole Swarm. After a certain number of iterations, the maximum (or minimum) value found as the Global Optimum is considered the optimal value for the function f.

3.1.1.2 Particle Swarm Optimization Algorithm

At the start of each iteration, each Particle has a position $x_i(t)$ and a velocity $v_i(t)$.

Update of Particles:

At the subsequent iteration, the update function will update the position and velocity of a Particle in accordance with the following rules:

$$v_i(t+1) = \alpha v_i(t) + \beta_1(x_i^{(local)}(t) - x_i(t)) + \beta_2(x^{(global)}(t) - x_i(t)) \tag{3.1.1}$$

$$x_i(t+1) = x_i(t) + v_i(t) \tag{3.1.2}$$

Parameter α represents an "inertia", which decreases over time, that is when t increases.

Parameters β_1 and β_2, are the weight assigned to the Local and Global "parts" respectively. They are usually chosen randomly at each step. They are called Cognitive Coefficient and Social Coefficient, respectively.

Parameter $x_i^{(local)}$ is the Local Memory of an individual (Particle), it represents the best coordinates in the Search Space visited by that individual. It is updated as follows:

$$x_i^{(local)} = x_i(\arg\max f(x_i(u))) \tag{3.1.3}$$

Parameter $x^{(global)}$ is the Global Memory of the Swarm, it represents the best coordinates in the Search Space visited by an individual in the Swarm, basically the best solution so far. It is updated as follows:

$$x^{(global)}(t) = x_j^{(local)}(t) \tag{3.1.4}$$

Whenever Local Optimum (for each Particle) and Global Optimum (for the whole Swarm) is found, the values of the two memories are updated.

Advantages of PSO Algorithm:

Differently from other more traditional optimization algorithms, PSO does not depend on the gradient of the objective function. Basically, unlike Gradient Descent, the movement of a Particle does not depend on which direction is "uphill" or "downhill", because the Particle is guided by Local Optimum and Global Optimum only.

This advantage makes PSO algorithms suitable for problems which objective functions are non-differentiable. Thus, it is an algorithm applicable to a wider range of optimization problems. Another advantage is that PSO is an "Embarrassingly Parallel" problem; a type of problem particularly easy to parallelize, as each particle can be updated in parallel.

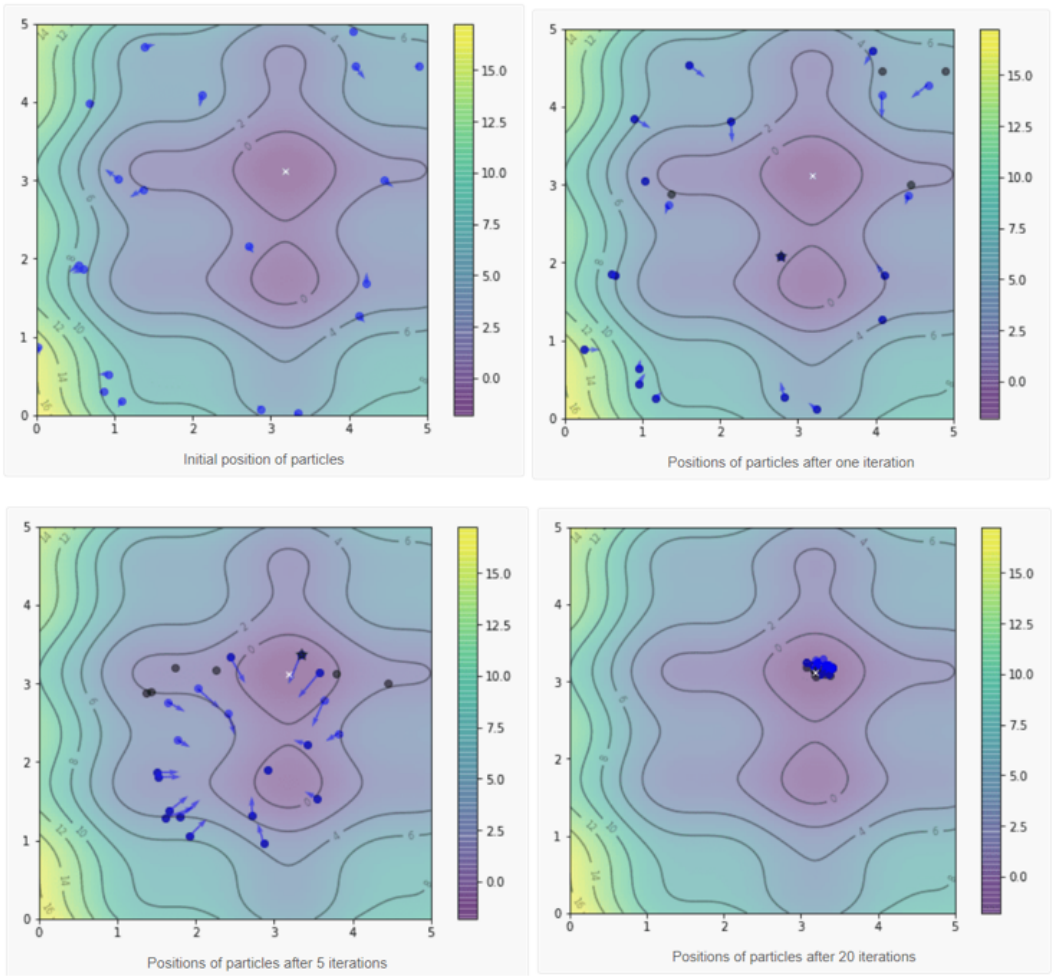

Figure 3.1.1: *Visual representation of the convergence of the Particles to the Global Optimum in the Search Space in a Particle Swarm Optimization Algorithm. Source: [24]*

Visual Example of PSO:

In the figure (Fig. 3.1.1), it can be observed how the particles progressively converge towards the Global Optimum, which represents the best solution found by the whole Swarm.

3.1.2 Components of Particle Swarm Optimization

Particle Swarm Optimization can have better results, be faster and cheaper compared to other methods. It is an easy problem to parallelize. Does not require the target function to be differentiable. It has few and not very complex hyperparameters.

In short, PSO has a large set of advantages; it is a modern solution to perform optimiza-

tion tasks. The final objective remains the same as for every optimization problem: to minimize (or maximize) a given function.

The three components of PSO algorithm are: Particle, Swarm and Optimization.

3.1.2.1 Particle

The Particle Swarm Optimization is inspired by the behaviour of flocks of birds. Therefore, the term "Particle", refers to a single individual in the Swarm.

Every Particle is defined by its Position and its Velocity in the Search Space. The Position in the Search Space allows for the evaluation of the values corresponding to that position, whereas the Velocity allows Particles to move stochastically in the Search Space, in search of new better positions, and thus solutions.

At the start of the optimization process, the positions of the Particles are defined randomly: random values of the Search Space are assigned to the Particles. The Velocity and direction of the Particles are also randomly generated.

The Particle in the PSO is thus an agent, the position this agent has in the Search Space is a potential solution, a candidate solution. Each Particle has Fitness values, which are evaluated by the Fitness Function, which is the function subject of the optimization. The Fitness Function evaluates a Particle's Positions, taking in input the values of the Search Space corresponding to that position.

3.1.2.2 Swarm

The Swarm is the Population of Particles in the optimization process, it represents the flock of birds.

PSO has similarities with Genetic Algorithms, but differently from them, there are no Evolution Operators to update the individual for the next generation. In PSO the next generation of individuals is an update of the former generation, which Position and Velocity are updated in order to improve the Fitness.

Inertia, Cognitive Intuition and Social Intuition:

After each iteration in the Search Space, the Velocity of each Particle is stochastically accelerated; consequently, also its Position will change.

The value the Velocity is going to be updated into is influenced by three factors: Inertia, Cognitive Intuition and Social Intuition. (Fig. 3.1.2)

Inertia is the tendency to keep the Velocity from the previous iteration. Cognitive Intuition is what makes the Particle accelerate toward the previous best Local Position, which is the best Position (corresponding to best Fitness) that Particle has achieved

$$P_i^{t+1} = P_i^t + V_i^{t+1}$$

$$V_i^{t+1} = wV_i^t + c_1 r_1 (P_{best(i)}^t - P_i^t) + c_2 r_2 (P_{bestglobal}^t - P_i^t)$$

Inertia | Cognitive (Personal) | Social (Global)

Figure 3.1.2: *Alternative update functions for the Velocity of the Particles in a Particle Swarm Optimization Algorithm, where the three components of the update function are represented, and additional parameters, c_1 and c_2 are introduced. Source: [25]*

so far. Social Intuition is what makes the Particle accelerate toward the previous best Global Position, which is the best Position (corresponding to best Fitness) the Particles in the Swarm have achieved so far.

3.1.2.3 Optimization

The Optimization component of PSO, consists in the actual process of updating the parameters, with the correlated setting of Hyperparameters. Inertia, Cognitive and Social coefficients have the function to control the levels of Exploitation and Exploration.

Exploitation means using the good solutions found so far in order to search for even better solutions in that mathematical neighbourhood. Exploration means to explore distant sections of the Search Space, in search of new information on the Space or potentially improving solutions.

At each iteration, both the Cognitive section and the Social section of the update formula are weighted by random terms [25] [26]. Basically, Cognitive acceleration and Social acceleration are stochastically adjusted by weights to make the update process more random and less deterministic. The possible values that the coefficients can assume are in Fig. 3.1.3.

The value of the Inertia coefficient defines the ability of the Swarm to change direction. Lower values of Inertia coefficient lead to better convergence; so low Inertia increases the exploitation of the best solutions. Higher values of Inertia coefficient increase the exploration around the best solutions. Values too high for Inertia, values > 1, cause divergence of the Particles [25].

The value of Cognitive coefficient defines the ability of the Swarm to be influenced by the personal solutions of each Particle. If the Cognitive coefficient is too high, then there will be no convergence, as each individual would be too focused on its optimal solution [25].

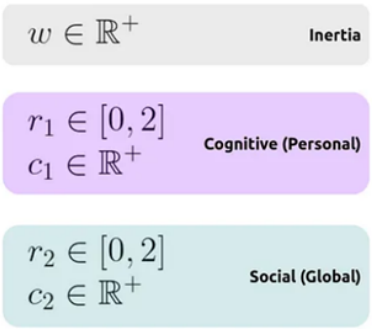

Figure 3.1.3: *Ranges of values for the Parameters of a Particle Swarm Optimization Algorithm. Source: [25]*

The value of Social coefficient defines the ability of the Swarm to be influenced by the best global solutions of the whole Swarm.

Auto Hyperparameters:

Inertia, Cognitive and Social coefficients are all three Hyperparameters for the optimization process of PSO.

Searching for the optimal values of their coefficient is a complex and expensive task, as it would require another optimization process. Moreover, the optimal value of these parameters changes during the optimization process, as the iterations go on.

Therefore, a satisfactory solution is to update the coefficients over the iterations [25]. Good recommended update formulas, coming from empirical studies, are the equations below.

The initial values should guarantee high exploration and more individuality, so high Inertia and Cognitive and low Social.

Toward the end of the optimization, values should guarantee high exploitation and convergence to the local optimum, so low Inertia and Cognitive and high Social.

$$w = 0.4\frac{(t-N)}{N^2} + 0.4 \tag{3.1.5}$$

$$c_1 = -3\frac{t}{N} + 3.5 \tag{3.1.6}$$

$$c_2 = +3\frac{t}{N} + 0.5 \tag{3.1.7}$$

3.1.3 Principles, Applications and Variants of PSO

This subsection, will examine: Principles of Swarm Intelligence, real-world applications of Particle Swarm Optimization, and PSO variants.

3.1.3.1 Principles of Particle Swarm Optimization

Particle Swarm Optimization has evolved a lot during its first experimental phase. While it was originally meant to simulate the behaviour of a flock of birds, the final form of the algorithm resembles more of a Swarm than a flock. For this reason, it took the name of Swarm Optimization.

The first researcher who talked about Swarm Intelligence, Millonas [27], defined five principles of Swarm Intelligence. Particle Swarm Optimization adheres to all five principles.

1. **Proximity Principle:** "The population should be able to perform simple space and time computations".
2. **Quality Principle:** "The population should be able to respond to quality factors in the environment". The PSO adheres to this principle because the population, the Swarm, tends to follow the two positions Local Optimum and Global Optimum, which are the environment's factors.
3. **Diverse Response Principle:** "The population should ensure enough diversity in its responses". The PSO adheres to this principle because the responses range from the Local Optimum of the Particle and the Global Optimum of the Swarm, ensuring diversity.
4. **Stability Principle:** "The population should not change its behaviour at each environmental change". The PSO adheres to this principle because the population only changes when the Global Optimum is updated and is therefore stable.
5. **Adaptability Principle:** "The population should be able to change its behaviour when it is worth the computational price". The PSO adheres to this principle because the population does change its behaviour when the Global Optimum is updated.

3.1.3.2 Applications of Particle Swarm Optimization

One of the main reasons why Particle Swarm Optimization is widely used as optimization technique, is that it is well-suited to a wide range of problems [28]. The advantage of PSO is that it has a small number of Hyperparameters to set, this allow the algorithm to be easily appliable to specific applications [28] [24] [23] [26].

- **Evolution of Neural Networks:** Particle Swarm Optimization can substitute traditional methods for the optimization of a Neural Network's weights. PSO is able to reach or outperform traditional approaches like Backpropagation. PSO can work so well with Neural Networks, that it can not only be used to optimize the networks' weights, but also their structure. PSO is effective for any network architecture [28].

- **Human Tremor Diagnosis:** PSO has been used in combination with Neural Networks for the diagnosis of Humar Tremor conditions, for example, Parkison's Disease [28].
- **End Milling Manufacturing:** PSO has been used in combination with Neural Networks for End Milling manufacturing [28].
- **Voltage Stability:** PSO has been used for dynamic power and voltage control in a Japanese electric establishment [28].
- **Determination of Battery State:** PSO has been used in combination with Neural Networks for estimating the state-of-charge of electrical or hybrid vehicles [28].

3.1.3.3 Variants of Particle Swarm Optimization

The Particle Swarm Optimization is a popular and effective optimization technique; therefore, many variants of the approach have been developed [25].

Variants primarily focus on adding evolutionary capabilities to PSO or improving performance with Hyperparameters.

- **Hybrid of Genetic Algorithm and PSO (GA-PSO):** implements the main aspects of GA approach, such as the capability of breeding and crossover.
- **Hybrid of Evolutionary Programming and PSO (EPSO):** implements the tournament selection in PSO, where the losing Particles change their position.
- **Adaptive PSO (APSO):** applies Fuzzy Logic to the Inertia coefficient. In addition, uses another PSO to perform the Hyperparameter Optimization for the first PSO.
- **Multi Objective PSO (MOPSO):** implements the concept of Pareto Dominance to determine which Particle should set the Global Optimum [26].
- **Discrete PSO (DPSO):** makes the performance of optimization better, for example using a mixed-search approach [26].

3.2 Custom Implementation of PSO

Particle Swarm Optimization, as described in the last section, is a powerful and successful Optimization algorithm, that can also be applied to HPO. In the last few years, many libraries for PSO have been developed, for instance, PySwarm, which is probably the most popular.

The problem with these libraries, as for the objectives of the experiments, is that they are hardly customizable, and often lack in explainability, making them less adaptable

to HPO. While this problem of explainability is potentially fixable, without being able to customize the original algorithm, the experiments would not be scientifically comparable between the ones run using Optuna's samplers.

Therefore, two potential solutions are proposed so as to apply PSO to an HPO problem: the first was to implement from scratch a PSO algorithm that follows the same style as Optuna so that the two would be comparable; the second was to implement an Optuna Sampler with PSO. The implementation of an Optuna Sampler with PSO will be discussed in the next section. In this section it will be examined the design of the PSO algorithm from scratch.

All the code mentioned in this section refers to the package `/experiments/PSO_experiment/backend/` of the code repository [22].

3.2.1 Implementation Details

The components of the algorithm are divided into five Python files: `PSO.py`, `pso_pruners.py`, `pso_runner.py`, `pso_train_utils.py` and `pso_utils.py`. These code files contain only what the algorithm needs to work, meaning that they do not contain the actual code of the experiments. In fact, the algorithm code is designed to mimic the functioning of Optuna, so that the same code to set up the Optuna experiments can be used to set up and perform experiments with this implementation of PSO. (Class Diagram in Fig. 3.2.1)

The contents and purpose of the main code files are the following:

- `PSO.py`: is the main file of the package, the actual core of the algorithm. It contains the `PSO` and `Particle` classes, together with some other private code.
- `pso_pruners.py`: contains the definition of the Pruners of the mini library. At the current state there is only one implemented Pruner which is the Median Pruner. It mimics the functioning of Optuna's Pruners.
- `pso_runner.py`: contains the definition of the `PSORunner` class, which is a wrapper of the optimization process that manages potential errors, logging, and saving of the results.
- `pso_train_utils.py`: contains the train loop function which should be used inside the objective function of the optimization process. It also contains some other private code.
- `pso_utils.py`: contains some constants and utility functions, in particular the one applied to encode and decode the value of the Hyperparameters from "internal" to "external" representations and vice-versa.

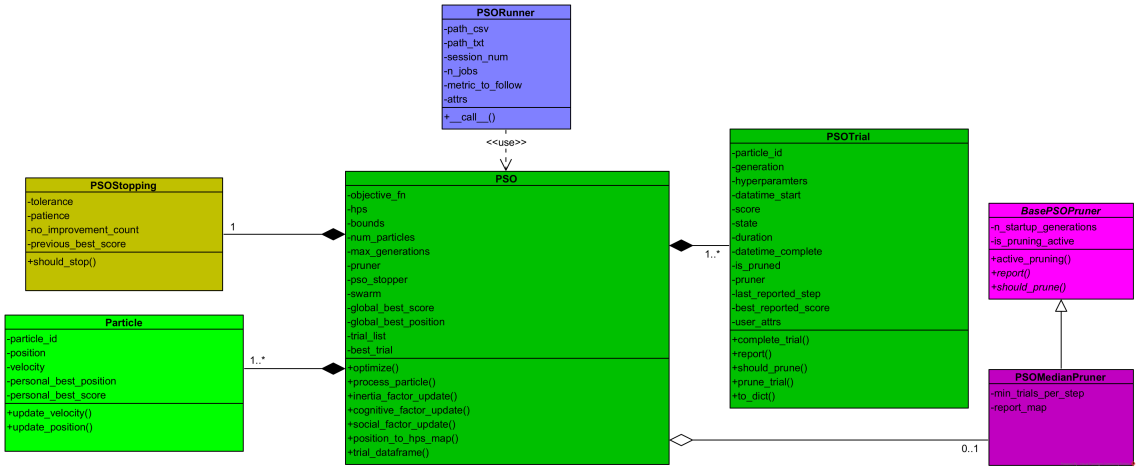

Figure 3.2.1: *Class Diagram of the custom implementation of the Particle Swarm Optimization with the related support code.*

3.2.2 Components of the PSO Algorithm

The true core of the algorithm, contained in `PSO.py`, consists of three fundamental components, which are: `Particle`, `PSOTrial` and `PSO`.

3.2.2.1 Particle

The `Particle` class is the representation of the concept of "Particle" in the original PSO algorithm.

In this specific implementation, the Particle has the following attributes: `paticle_id`, `position`, `velocity`, `personal_best_position`, `personal_best_score`. The position and the velocity of the particle are initialized with randomly generated arrays.

The particle has two methods, which are `update_velocity()` and `update_position()`. The version of the strategy for the velocity update is the one with the Cognitive and the Social coefficients (c_1 and c_2). The random coefficients (r_1 and r_2) are chosen randomly, whereas, the other two coefficients, together with the Inertia coefficient (w), are passed as an input. The strategy for the position update is the standard one, with the exception that values which go outside the Search Space are clipped (approximated to the nearest bound value).

3.2.2.2 PSOTrial

The `PSOTrial` class, as the name suggests, is a representation of a trial during the optimization. Its functioning is inspired by the `Trial` object of Optuna. Its main objective is to carry metadata regarding each trial of the optimization process, so as to the results can be displayed the same way they are in Optuna.

The `PSOTrial` object has the following attributes: `particle_id`, `generation`, `hyperparameters`, `datetime_start`, `datetime_complete`, `duration`, `score`, `state`, `user_attrs`, `is_pruned`, `pruner`, `last_reported_step`, `best_reported_score`. The last four attributes are related to the pruning system. The `user_attrs` attribute is a map object which allows the user to set custom metadata in the trial, like it can be done in Optuna. The other attributes are self-explainatory.

The methods realize functionalities such as completion of a trial, conversion to map, setters, and functions related to pruning.

3.2.2.3 PSO

The `PSO` class is the representation of the actual PSO algorithm. It is designed to be equivalent to a `Study` object of Optuna. One instance of the `PSO` class carries on a whole optimization process, in the same way as an instance of Optuna's `Study` instance.

The attributes of the `PSO` class are the following:

- `objective_function`: the objective function of the HPO process, to be passed at the initialization of the object just like in Optuna.
- `bounds`: is a 2D array which contains the lower bounds and the upper bounds of each non-categorical hyperparameter.
- `hps`: contains the mapping between the hyperparameter name and the index inside the inner arrays of bounds.
- `num_particles`: is a hyperparameter of the algorithm itself; it represents the number of Particles in the Swarm.
- `max_generations`: is another hyperparameter of the algorithm itself; it represents the maximum number of iterations (Generations) of the algorithm.
- `pruner`: the Pruner object to use in the algorithm.
- `pso_stopper`: must be an `PSOStopper` object, which is another secondary class defined in the file. It Early-Stops the process according to a specified Tolerance and Patience.
- `swarm`: is the representation of the concept of Swarm. Rather than creating a specific class for it, it simply is a list of Particles (list of `Particle`).
- `global_best_score`: the best score at any given point in time.
- `global_best_position`: array which values represent the coordinates in the Search Space for the best position at any given point in time.
- `trials_list`: list of all the `PSOTrial` objects.
- `best_trial`: initialized at the end of the optimization, stores the best trial.

The methods of the `PSO` class, excluding the private or less important methods, are the following: `optimize()`, `inertia_factor_update()`, `cognitive_factor_update()`, `social_factor_update()`. The three update methods are implemented using the formulas explained in the Particle Swarm Optimization section (Sec. 3.1.1.2).

The `optimize()` method is where the actual PSO algorithm takes place. Here is where the optimization loop happens. The first level loop iterates over the maximum number of generations, then the inner loop iterates over each particle of the swarm. After the particle is evaluated during a determined generation, and thus in a determined position, two checks are made to determine if the current result is better than the local and global better results. Finally, at the end of a generation, each particle is updated using its update methods, with the values for w, c_1 and c_2 obtained from the three update methods mentioned earlier. At the end of the whole loop, the method ends, returning the best global hyperparameters and the best global score.

3.3 Enhancing Optuna Optimization

Hyperparameter Optimization is expensive, and tools like Optuna not only exist to make the process more efficient but also to simplify the setup of the optimization. Nevertheless, especially when the experiments grow in complexity, even tools like Optuna start to show weaknesses and robustness problems.

Therefore, it is necessary to use Optuna to its full potential, gaining advantage from its additional abilities, such as the Storage. Moreover, an infrastructure beyond Optuna is also necessary to handle error throwing and to make persistent the data that Optuna Storage does not memorize.

These extras are not always easy to setup, or at least require a setup for each Study. This, added to the already present normal setup of the Study, with the creation of the object and the launch of the optimization, makes the Study initialization process very heavy. This would be perfectly fine in normal circumstances, but the goal of the proposed experiments, is to compare the different optimization techniques and sampling algorithms, thus, the previously described workflow should be repeated for each Study.

The solution is to abstract some functionalities of Optuna, basically wrapping them, so as to reduce the quantity of code to write and improve the performance and reliability of the processes. (Object Diagram in Fig. 3.3.1)

All the code mentioned in this section refers to the package `/utils/` of the code repository [22].

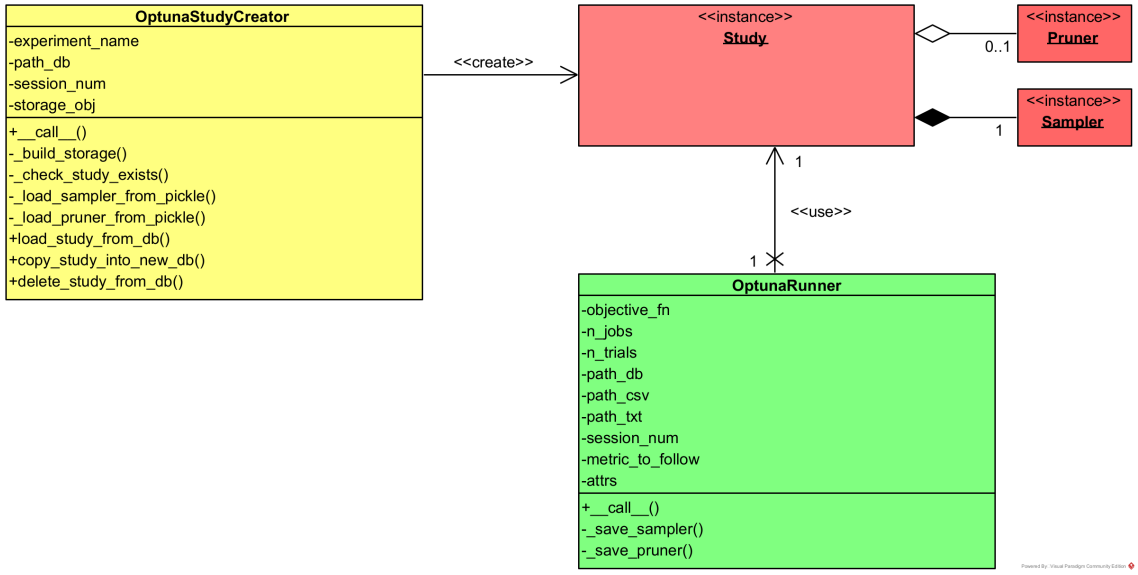

Figure 3.3.1: *Object Diagram of the "wrapping", or enhancement, of Optuna functionalities.*

3.3.1 Optuna Wrapping

The two main concerns relative to the abstraction of Optuna's functionalities are the initialization of the Study and the running of the optimization. The proposed infrastructure, which wraps those two functionalities, has various objectives: to implement persistency (for both Optuna and additional metadata such as logs), to handle errors, to save the execution state of samplers and pruners (which Optuna does not do natively), to log and save the results in various formats, and other capabilities (like copying a Study to another DB, loading a Study from a DB, deleting a Study).

As a result, by abstracting the process of creation and run of the optimization, the code is totally reusable for each Sampler, therefore making each singular experiment comparable to one another.

The main files of the wrapping are `optuna_study_creator.py` and `optuna_runner.py`.

3.3.1.1 Optuna Study Creator

Contained in the file `optuna_study_creator.py`, the class `OptunaStudyCreator` abstracts the process of creation of one single Study representing the optimization experiment.

After initializing the object with values which will be common for all the studies, by calling it, it is possible to create new `Study` objects. In particular the creator checks if the DB is present at all, checks if the study is already present in the DB, initializes the DB, initializes the Study representation in the DB, defines the storage object (which

is the connection between the Study and its database). In case the Study needs to be loaded, the creator also checks if there are pruners or samplers' states saved and eventually loads them into the optimization process.

The class also contains methods to copy a Study to another DB, load a study from another DB and delete a Study from a DB.

3.3.1.2 Optuna Runner

Contained in the file `optuna_runner.py`, the class `OptunaRunner` abstracts the process of optimization of one single Study representing the optimization experiment.

After initializing the object with values which will be common for all the studies, by calling it and passing the `Study` as input, it is possible to run the Study, equivalently to the `optimize()` method of Optuna. The runner checks if there is any log file already present, creates the log file, runs the optimization, catches exceptions, saves the execution state of the Sampler and the Pruner of the Study, logs and saves the results of the optimization experiment.

3.3.1.3 Other Optuna-Related Code

As mentioned earlier, the proposed experiments create and run multiple optimizations, therefore, in addition to the strategies explained so far, another part of the code which needs to be as much standardized and abstract as possible is the Objective Function.

The following files in the `/utils/` package contain code that helps with this.

- `train_loop.py`: contains the function that executes the training of the model object of the trial. Inside the loop are also defined the instructions related to pruning, early stopping, logging, and data persistence.
- `early_stopper.py`: contains the definition of the `EarlyStopper` object, utilized to early-stop training loops within a trial which are no longer improving.
- `regularizer.py`: contains the definition of the `Regularizer` object, utilized for the Regularization in the experiments.
- `logger.py`: contains the definition of the `Logger` object, which is used throughout the whole optimization process to memorize debug information, errors, warnings, and logging information.
- `file_name_builder.py`: contains utils functions to create folders and filenames for the files of logging or persistence in general (such else results files).
- `build_dateset.py`: contains functions that abstract the creation of the dataset, regardless of the particular experiment.

- `build_dataloader.py`: contains functions that abstract the creation of the data loaders, regardless of the particular experiment.

- `model_utils.py`: arguably the most important of this list, contains various functions, called in the objective function, which allow to initialize the components of the optimization (Loss Function, Model HPs, Training HPs) using the converted values of HPs sampled by Optuna.

3.3.2 Custom Optuna PSO Sampler

As mentioned in the section of the Custom PSO Implementation (Sec. 3.2), one alternative approach to HPO using PSO was to integrate a PSO algorithm in Optuna as a Sampler.

Optuna allows developers to define new Samplers. In order to do this, the custom sampler should inherit from the class `BaseSampler` of Optuna and override some methods.

The code described in this section refers to the `/utils/optuna_utils/pso_sampler.py` file of the code repository [22].

3.3.2.1 Initialization

The Sampler requires two initialization parameters, which are the number of particles, and the number of generations. Even if, in the current state of the implementation, there are no integrity checks, these values should be compatible with the number of trials to execute, which is a parameter of the Study running function of Optuna.

As soon as the first trial starts, the Swarm is initialized; it is a list of Particles, where a Particle is defined in the same exact way described in the section of Custom Implementation of PSO (Sec. 3.2).

Until a trial is completed or pruned, Optuna calls the `sample_indipendent()` method, which samples the value for the hyperparameters randomly with a `RandomSampler`.

After the completion of the first trial, the Search Space is also initialized. The Search Space cannot be initialized earlier by Optuna because the current possible value ranges of the Hyperparameters are unknown to Optuna before running the Objective Function at least once.

The Sequence Diagram of the `BaseSampler` (Fig. 3.3.1) allows to understand how the `PSOSampler` interacts with the rest of the Optuna components and executes the sampling.

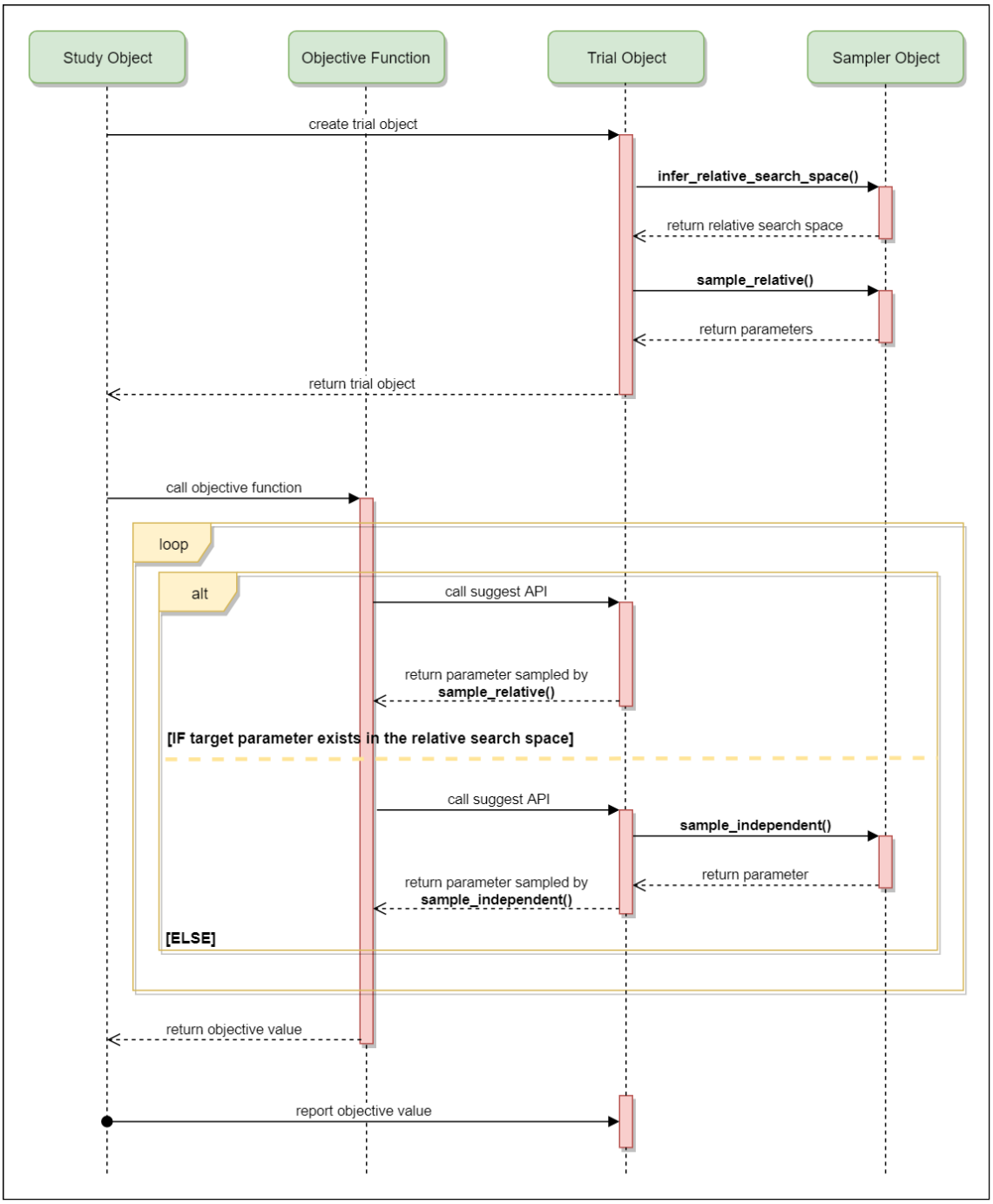

Figure 3.3.2: *Sequence Diagram of the interaction between Optuna's Samplers and the rest of the Optuna's components. Through the diagram, is possibile to understand how a sampler in Optuna behaves at the start of a Trial. Source: [20]*

3.3.2.2 Sampling

At the start of each Trial, the `before_trial()` method is called. There, depending on its number, a specific Particle and the current generation are assigned to the Trial.

The Sampling of each candidate tuple of Hyperparameters happens through the `sample_relative()` method, which after checking what Particle the current Trial refers to, gets the position of the Particle and converts its coordinates into HPs values.

3.3.2.3 After Trial

At the end of each trial, the `after_trial()` method is called. The Particle associated with the Trial is updated; the values for w, c_1 and c_2 are obtained in the same way described in the section on Custom Implementation of PSO (Sec. 3.2). Local and Global best positions and scores are also checked and eventually updated.

3.4 Applied Neural Network Models

The type of Machine Learning model chosen for the experiment is Neural Networks. The chosen library to implement the NN Models and the related support code (such as Loss Function, Train Loop) is PyTorch, which is the most popular Deep Learning library at the moment [29]. In the experiments, everything related to the architecture and to the training and evaluation of the models is implemented using PyTorch.

3.4.1 Models

In the experiments, two models have been used: MLP and Lawin.

MLP, or Multi-Layer Perceptron, is the simplest type of Deep Learning Model. The implementation that was utilized is not the standard one of PyTorch but is a custom one, developed using PyTorch. The MLP is applied as the ML model for all the experiments except the last one.

Lawin is a complex DL model specialized in Semantic Segmentation, it is applied only to the Weed Map experiment.

3.4.1.1 MLP

This implementation of the MLP simply follows the standard characteristics of the basic form of MLP (Fig. 3.4.1). Its architecture is represented by a list of values, where the length of the list is the Depth of the Hidden Layer, and the value in the list is the Width of the Layer at the corresponding index. Both the Architecture and the Activation Function are initialization inputs.

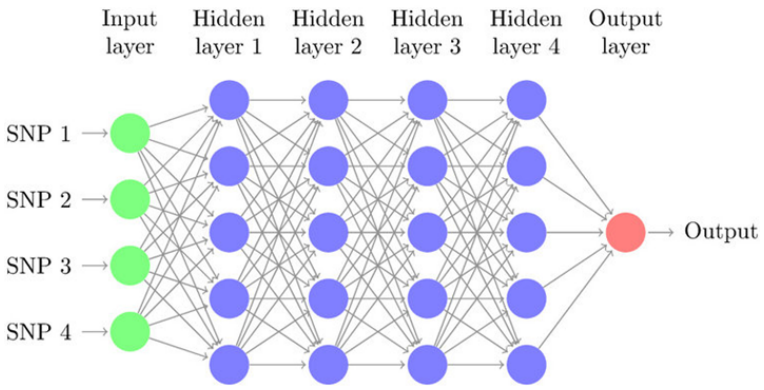

Figure 3.4.1: *Visual representation of a MLP Neural Network. In this specific example, the input layer consists of four neurons, the output layer of just one, and each hidden layer of five; the width of the hidden part of the network is in this case four. Source: Google Images*

This implementation of MLP refers to the file `/utils/model/MLP.py` of the code repository [22].

3.4.1.2 Lawin

Lawin is a complex Deep Learning model, in particular is a Vision Transformer, specialized in Semantic Segmentation and applied to Drone Vision problems. Lawin was developed for this article [30], and its implementation can be found in the relative repository [31].

Lawin consists of an Encoder and a Decoder (Fig. 3.4.2). The encoder is a Mix Transformer (MiT), which can generate CNN-like multi-level features with different resolutions, providing a feature map for each Transformer block as output. The decoder uses a Large Window Attention Spatial Pyramid Pooling (LawinASPP), consisting of five parallel branches, a pooling layer, a shortcut connection, and three large window attentions with different context sizes.

Variants of Lawin are: Double Lawin and Split Lawin. Those variants are mentioned in this article [30] but are not utilized in any of the proposed experiments.

3.4.2 Loss Functions

In Neural Networks, during the training process, the Loss Function calculates the distance between the prediction of the model and the actual target value.

In the experiments, two loss functions have been used, which are Cross Entropy and Focal. For the Weed Map experiment only FocalLoss has been used, whereas both (or sometimes only CrossEntropyLoss) have been used for all other experiments.

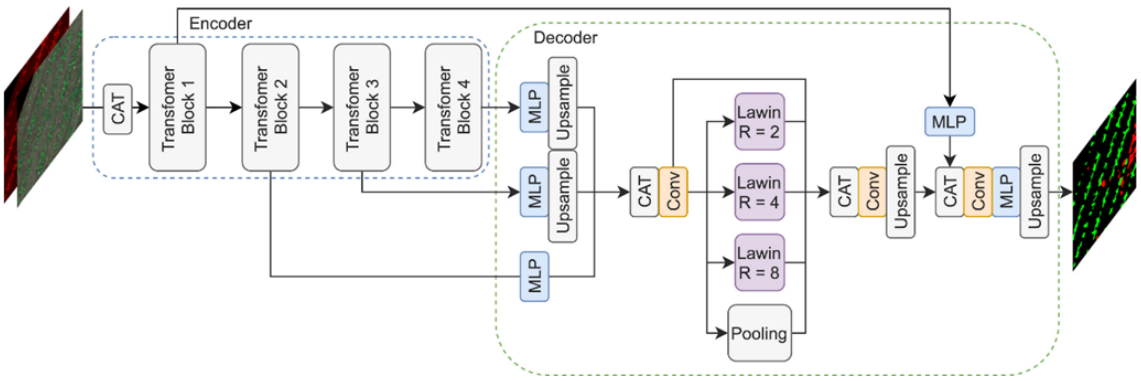

Figure 3.4.2: *Architecture of Lawin. Source: [30]*

`CrossEntropy` is the implementation from PyTorch, the one inside the package `nn`. `FocalLoss` is implemented with PyTorch.

3.4.2.1 Cross Entropy Loss

Cross Entropy Loss, a loss function principally applied in classification problems, measures the difference between two probability distributions: the true distribution (actual labels) and the predicted distribution (model outputs). Basically, it determines how far the model's predictions are from the labels (the targets).

3.4.2.2 Focal Loss

Focal Loss is an extension of Cross Entropy Loss. This particular implementation of Focal Loss is from the same article and repository mentioned in the previous sections about Lawin [30] [31]. Focal Loss is calculated as follows:

$$FL = -w_c(1 - f(x)_c)log(f(x)_c) \tag{3.4.1}$$

In the formula, $f(x)_c$ is the probability of the true class predicted by the model, and w_c is the corresponding class weight. An interpretation of this formula is that different weights are given to different classes of the classification task. This is to address the class imbalance, particularly in scenarios where some classes are significantly more frequent than others.

A more detailed explanation of Focal Loss is in the earlier mentioned article [30]. The original form of Focal Loss was presented in this other article [32].

3.4.3 Training and Evaluation

The functions to train and evaluate the models are also written using PyTorch.

All the code mentioned in this section refers to the package `/utils/training` of the

code repository [22]. The three illustrated files are `eval_step.py`, `train_step.py`, and `train_loop.py`.

3.4.3.1 Training Step

In the `train_step()` function, the PyTorch model is set in Training Mode, and the backward step is executed, calculating the gradients. Basically, this Training Step is the training loop within a single Epoch.

3.4.3.2 Evaluation Step

In the `eval_step()` function the PyTorch model is set in Evaluation Mode, then predictions are made using the validation or test data loaders and compared with the target values. The performance is evaluated using a specific Metric. In order for this code to be as abstract as possible, a class called `MetricWrapper` (defined in the `/utils/metrics` package) was implemented, which abstracts the methods of initialization, computing, and value return of the metrics.

All four main evaluation metrics are calculated (Accuracy, Precision, Recall, F1), but only one is considered toward the final score of the evaluation. The chosen metric is F1 for the Weed Map experiment and Accuracy for all other experiments.

3.4.3.3 Training Loop

In the `full_train_loop()` function, the PyTorch model is fully trained, iterating over the maximum number of epochs, and executing `train_step()` and `eval_step()` at each iteration. The function also contains instructions related to pruning, early stopping, logging, and data persistence.

3.4.4 Regularization

Regularization is a technique used to prevent overfitting in Neural Networks. More generally, it is a technique that can be used in multi-objective optimization, to penalize certain characteristics of a candidate solution by lowering its score.

In the proposed experiments, the Regularization technique has the goal of penalizing the complexity of the model, in terms of the width and depth of the hidden layers. Basically, a penalty term is applied referring to the complexity of the model's architecture.

In the experiments, as all of them share a secondary objective to reduce the complexity of the found solutions, a penalization toward architecture complexity was applied. Whenever "Score" is mentioned, it refers to the regularized score, rather than the specific metric score (Accuracy or F1). Which is calculated as follows:

$$Score = ms - \lambda \cdot complexity \tag{3.4.2}$$

In the formula: ms is the metric score (could be Accuracy or F1), λ (`lambda`) is the Regularization coefficient (which is passed as input) and *complexity* is the complexity of the model, which is calculated as the sum of the number of neurons in each hidden layer.

The Regularization was implemented with a `Regularizer` class defined in the `/utils/optimization/regularizer.py` file of the code repository [22].

3.4.5 Other Training-Related Objects

Inside the `/utils/model/model_utils.py` file of the code repository [22] are defined other aspects of the training process of the model, such as the initialization of Activation Functions and Optimizers.

The possible Activation Functions are Sigmoid, ReLU and Softmax. The `get_activation_fn()` function takes as an input the string sampled from Optuna representing one Activation Function and returns the initialized corresponding object using the PyTorch implementation.

The possible Optimizers are SGD (Stochastic Gradient Descent) and ADAM. The `get_optimizer()` function takes as an input the string sampled from Optuna representing one Optimizer and returns the initialized corresponding object using the PyTorch implementation.

Chapter 4

Results

In this chapter, the proposed experiments are presented and explained. For each experiment, the results are shown and discussed. There is a total of five proposed experiments:

- **Experiment 1 - Preliminary Experiment:** aims to demonstrate the necessity of Hyperparameter Optimization (HPO) and to familiarize with the concepts of HPO.
- **Experiment 2 - Comparison of Optuna's Samplers on MNIST:** aims to evaluate the performance of different Hyperparameter Optimization algorithms, specifically Optuna's Samplers, on the MNIST dataset.
- **Experiment 3 - Comparison between a Custom PSO implementation and Optuna:** aims to compare the performance of a custom implementation of Particle Swarm Optimization (PSO) with Optuna's Samplers, on the MNIST dataset.
- **Experiment 4 - Comparison between a PSO Sampler and Optuna's Samplers:** aims to compare the performance of a PSO Sampler (made using Optuna as a base) with the other Optuna's Samplers, on the MNIST dataset.
- **Experiment 5 - Hyperparameter Optimization on Weed Map Dataset:** aims to perform Hyperparameter Optimization on the Weed Map Dataset, using Optuna.

Experiments Source Code:

All the source code of the experiments is available on the same repository [22] mentioned in the previous chapter (Chapter 3), specifically in the `/experiments/` package. In particular, the Weed Map Experiment (Experiment 5) is in the file `/weed_mapping_experiment/Weed_Mapping_Optimization.ipynb`, while the other experiments were run using `/MNIST_experiment/MNIST_Optimization.ipynb`.

4.1 Experiment 1 - Preliminary Experiment

This Preliminary Experiment has a double goal. The first is to familiarize with the concepts of HPO from a physical implementation perspective. The second reason is to establish, if ever needed, the necessity of Hyperparameter Optimization; to demonstrate that HPO can actually improve the final performance of a ML experiment by finding the best Hyperparameters to train the model subject of the experiment with.

Note that in this preliminary experiment, everything was developed with simple vanilla Python, without the use of the infrastructure described in the previous chapter (Chapter 3).

System used for the Experiment:

The hardware that was utilized to run this experiment is the following:

- CPU: Intel Core i5-8265U CPU @ 1.60GHz (4-core, 8 thread)
- GPU: NVIDIA GeForce MX 110 (2GB VRAM)
- RAM: 8GB (DDR4)
- Disk: SSD
- OS: Windows 11

4.1.1 Preparation of the Experiment

Given the first goal of the experiment, the idea is to realize a simple implementation of the most traditional of HPO algorithms, which is Grid Search.

The first part of the experiment consists in recreating another ML experiment which does not apply HPO and evaluating its results. Successively, the hyperparameters of that same ML model will be optimized with the custom Grid Search.

The experiment chosen for comparison is the one proposed by the book "Dive Into Deep Learning" [2] (chapter 4 of the book). Some modifications have been made to this experiment: a simpler version of the proposed dataset is used; the Multi-Layer Perceptron (MLP) considered is implemented with vanilla Python rather than PyTorch.

4.1.1.1 Dataset of the Experiment:

The Dataset considered for the experiment is a simplified version of MNIST, specifically, the version included in scikit-learn [19], from the function `load_digits()` of the `dataset` module. (An explanation of the complete MNIST dataset is in the next experiment; for this preliminary experiment it is not necessary to understand the dataset).

4.1.1.2 Model and Hyperparameters of the Experiment:

As introduced earlier, the MLP model utilized is not the same as the one from the book. This is not detrimental to the scientific value of the experiment, as the considered hyperparameters and configuration of the model were the same as those proposed in the book, so that a comparison would still be possible.

4.1.1.3 Implementation of Grid Search:

The proposed implementation of Grid Search is completely made from scratch. The algorithm takes as input the model, the dataset, and the grid of hyperparameters; the grid is a map object, where the keys are strings that represent the name of the hyperparameter, and the values are lists which contain the values that the corresponding hyperparameter can be.

At the start of the algorithm, a Cartesian Product is applied to the lists inside the `param_grid`, basically generating all the trials that are going to be evaluated. For each trial, the model is initialized with the hyperparameters of that trial and evaluated; the results are saved in a variable. After having completed all the trials, the algorithm returns as output the variable containing all the results, from which the best trial can be obtained.

Two versions of this algorithm have been developed, one without parallelism and one with parallelism. There is no difference between the two, the only one is that the second allows to execute multiple trials on multiple threads.

4.1.2 Execution of the Experiment

The execution of the experiment is divided into three short parts: run and evaluation without hyperparameter optimization; run and evaluation with hyperparameter optimization; comparison of results.

The set of hyperparameter utilized by the book for the experiment are the following (Tab. 4.1.1). Training and evaluating the model with these hyperparameters lead to results of 99% accuracy on the Training Set and 97% on the Test Set. The custom Grid Search algorithm is executed with the `param_grid` in the figure (Fig. 4.1.1) as input.

The Grid Search algorithm executed a total of 576 trials. After the completion of the Optimization, there are three best results (with the same accuracy), which are in the figure (Fig. 4.1.2).

These best results have an accuracy of 100% on the Training Set and 98.6% on the Test Set. Moreover, there are more than 20 trials with better accuracy than the configuration proposed by the book.

Table 4.1.1: *Initial hyperparameters of Experiment 1, as proposed by the book.*

Hyperparamter	Value
Activation Input	Sigmoid
Activation Output	Softmax
Batch Size	256
Learning Rate	0.1
Hidden Size	10
Epochs	10000

```
param_grid = {
    'hidden_size': [32, 64, 128, 256],
    'activation_hidden': ['sigmoid', 'relu'],
    'activation_output': ['sigmoid', 'softmax'],
    'learning_rate': [0.01, 0.1, 0.5],
    'epochs': [101, 501, 1001],
    'batch_size': [16, 32, 64, 128],
}
```

Figure 4.1.1: *The Param Grid utilized for the Custom Grid Search Experiment.*

4.1.3 Discussion of the Results

This simple experiment with its results outlines how hyperparameter optimization, even when using a simple implementation of the weakest algorithm, could lead to a significant improvement in the performance of the trained model.

The gained 1.6% accuracy on the Test Set may seem like a little improvement, but is remarkable, especially when the dataset is composed of a very large number of observations.

These results, additionally, confirm another aspect of HPO, which is the ability to improve the performance of an experiment without interfering with the model or the dataset.

hidden_size	activation_hidden	activation_ou…	learning_rate	epochs	batch_size	accuracy
256	relu	sigmoid	0.50	1001	32	0.986111
64	sigmoid	sigmoid	0.50	501	16	0.986111
128	relu	sigmoid	0.50	1001	16	0.986111

Figure 4.1.2: *The three best results of the Custom Grid Search Experiment.*

4.2 Experiment 2 - Comparison of Optuna's Samplers on MNIST

The objective of this experiment is to evaluate the performance of different Hyperparameter Optimization algorithms so as to gain general knowledge about the capacity of each in preparation for subsequent, more complex experiments.

In order to make the results and the methodology as much scientifically comparable as possible, rather than implementing various Samplers from scratch individually, a library for HPO is used.

The chosen library is Optuna, which, as described in previous sections, includes a few Samplers already implemented, with the same style and same preconditions, making each optimization perfectly comparable to one another.

As the idea is to have results that are as much general as possible, and because such "optimization of optimizations" is inevitably enormously expensive, a simple dataset is considered, which is MNIST.

System used for the Experiment:

The hardware that was utilized to run this experiment is the following:

- CPU: Intel Core i7-10700K CPU @ 3.80GHz
- GPU: NVIDIA GeForce RTX 3080 (10GB VRAM)
- RAM: 16GB (DDR5)
- Disk: SSD
- OS: Linux (Ubuntu)

4.2.1 MNIST Dataset

The MNIST Dataset is a database of images of handwritten digits [33]. MNIST is the ideal dataset for learners and for small experiments regarding the field of pattern recognition, is basically a benchmarking dataset for Image Classification.

Each image of the MNIST Dataset represents a digit from 0 to 9, making it ideal for benchmarking simple classification models for images (Fig. 4.2.1).

4.2.1.1 Structure of MNIST Dataset

The MNIST Dataset is composed of a total of 70000 images, of which 60000 are part of the training set and 10000 are part of the test set [33]. The total weight of MNIST is 11.6MB when compressed, 52.3MB normally. Each image is 28x28 pixels, is in black and white, and weighs 784 bytes.

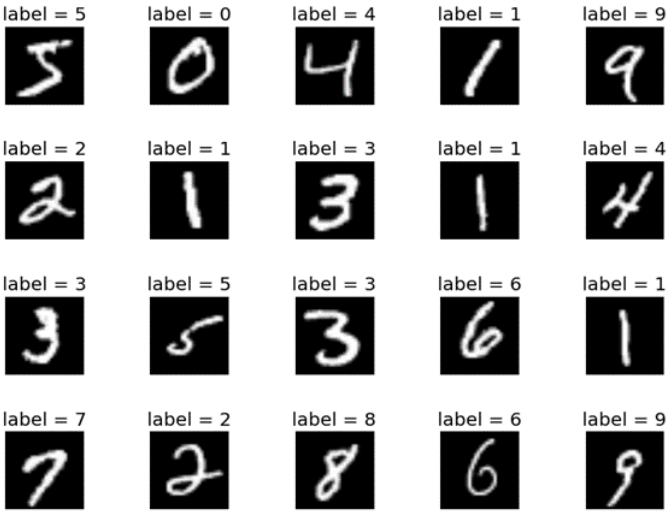

Figure 4.2.1: *Some examples of images in the original MNIST dataset. Source: [33]*

NIST Dataset:

MNIST is a newer version of an old dataset called NIST, in which the division was between SD-1 (Special Database 1) and SD-3 (Special Database 3); where SD-3 was the training set, and SD-1 the test set. The images of SD-3 were much clearer and easier to recognize.

MNIST is designed so that half of its training images and half of its test images come from SD-1 and the remaining half from SD-3. This means that half of the images of MNIST are easier to recognize, the other half are more difficult.

4.2.1.2 Variants of MNIST Dataset

Given its extreme simplicity to both use and understand, MNIST has inspired numerous variants. Some of the most famous variants are the following:

- **Fashion-MNIST:** replaces digits with images of clothing items to test models on more complex visual patterns (Fig. 4.2.2).
- **EMNIST (Extended MNIST):** includes, in addition to digits, also letters, to enhance the diversity of the dataset.
- **KMNIST:** includes images of Japanese Kanji characters, making the classification task much more complex than the version on the Latin alphabet.
- **NoiseMNIST:** is a version of MNIST where Gaussian noise is added to the images, it is useful to test the robustness of the model against noisy data.

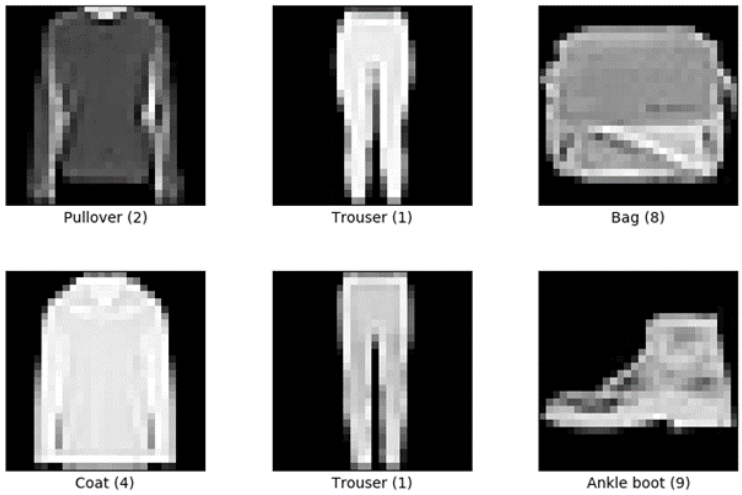

Figure 4.2.2: *Some examples of images in the Fashion-MNIST dataset. Source: Google Images*

- **Rotated MNIST:** is a version of MNIST where the images are rotated by a random angle, to test the robustness of the model against rotated data.

4.2.2 Preparation of the Experiment

The preparation for the experiment is simple: using the code infrastructure described in the Methodology chapter, specifically the section on Optuna Enhancing (Sec. 3.3), an Optuna's `Study` object is created and run.

The MNIST dataset was imported through the `MNIST` package of Pytorch, included in torchvision, and then loaded with PyTorch data loaders.

4.2.2.1 Setup of the Studies

The Samplers of Optuna subject of this experiment are the following: `RandomSampler`, `TPESampler`, `CmaEsSampler`, `GPSampler`, `NSGAIISampler`, `PartiallyFixedSampler`, `QMCSampler`. For each of the Samplers, an Optuna `Study` is created. All the samplers are initialized with their default input values.

The Pruner utilized for each Study is the Hyperband (`HyperbandPruner` object in Optuna), initialized with its default input values.

4.2.2.2 Setup of the Objective Function

The objective function, obviously the same for each Study, is divided into three sections: Declaration of the Hyperparameters to optimize and their value ranges; Initialization of the model and related objects; Training and Evaluation of the model.

At the execution of each trial, for each hyperparameter, a value is sampled by Optuna's Sampler, then those values are used to initialize the model and its related objects, training included, and the training loop can start. The value then returned will be the score.

The Hyperparameters declared in the objective function, which form the Search Space of the Study, for this experiment were the following (Tab. 4.2.1).

Table 4.2.1: *Search Space of Experiment 2: list of hyperparameters and their values ranges, with the step. Where Step is ∼ if the range is continuous.*

Hyperparamter	**Range**	**Step**
Activation Function	{ReLU, tanh, sigmoid}	-
Batch Size	[16, 64]	16
Epochs	10	-
Hidden Layer Width {i}	[8, 128]	8
Hidden Layer Depth	[1, 3]	1
Learning Rate	$[10^{-5}, 10^{-1}]$	∼
Loss Function	{CrossEntropy, Focal}	-
Optimizer	{SGD, Adam}	-

Other parameters, related to the Optimization process, basically hyperparameters of the HPO process itself (could be called "Hyper-Hyperparamters"), which were used are the following (Tab. 4.2.2).

Table 4.2.2: *External optimization parameters of Experiment 2, basically hyperparameters of the optimization process, with their values.*

Group	**Parameter**	**Value**
Regularizer	labda	0.4
EarlyStopper	patience	5

4.2.3 Execution of the Experiment

The execution of each Study is handled by an `OptunaRunner` object, which at the end of the optimization process saves the results on a csv file.

The number of Trials executed for each Study was 128.

In the reporting of the results, the information regarding what the best hyperparameters' values are will be ignored, because it is not the focus of the experiment, which is to identify how well performing the various techniques are.

For the experiment, each Study has been executed around six times (some had more executions than others); the reported results are therefore a mean of the multiple results from the multiple executions.

4.2.3.1 RandomSampler

The best score achieved by the `RandomSampler` is 91.7%; there are 29 trials with a score over 80% and 3 with a score over 90%. The Accuracy of these top trials is around 95%-97%.

In terms of Accuracy, the best score is 98% and the best results are around 97%, but those results have lower regularized scores.

The results of the `RandomSampler` are useful to compare the results of the other samplers, to see if there is an actual advantage in using them for these experiments, or if they are just equivalent to the random approach.

4.2.3.2 TPESampler

The best score achieved by the `TPESampler` is 94%, which is the best score of the whole experiment; there are 101 trials with a score over 80%; 60 with a score over 90%; 13 with a score over 93%. The Accuracy of these top trials is around 94%-96%.

In terms of Accuracy, the best scores are all close to 98%, but all those best scores are also the worst in terms of regularized score.

Talking about the network architectures, the best scores were achieved with a single hidden layer architecture, other good scores were achieved even with a linear architecture, or with a two-hidden-layer architecture where the total number of neurons was minimally higher than the single layer one.

4.2.3.3 CmaEsSampler

After a total of five tries, the execution of the studies never terminated without errors. It is likely that the setup values of the experiment are not compatible with the default initialization values of this sampler. Therefore, this Sampler will be discarded.

4.2.3.4 GPSampler

The best scores achieved by `GPSampler`, both in terms of Accuracy and in terms of regularized score, are basically equivalent to the results of `RandomSampler`. In order to give more possibilities to the sampler, various initialization values were tried, none of

which changed the results. It is not completely clear why this sampler produces results equivalent to those of `RandomSampler`, this will be further discussed in the Discussion section.

4.2.3.5 NSGAIISampler

The best scores achieved by `NSGAIISampler`, both in terms of Accuracy and in terms of regularized score, are basically equivalent to the results of `RandomSampler`. In order to give more possibilities to the sampler, various initialization values were tried, none of which changed the results. It is not completely clear why this sampler produces results equivalent to those of `RandomSampler`, this will be further discussed in the Discussion section.

4.2.3.6 PartiallyFixedSampler

The `PartiallyFixedSampler` was failing in the first execution because, as the name suggests, it is a type of sampler that requires some hyperparameters of the defined Search Space to be fixed.

The choice was not to further continue using this sampler, as the objective of the experiment is to evaluate the sampling technique rather than to actually find the best hyperparameters' values, or correlations in the Search Space. Therefore, this Sampler will be discarded.

4.2.3.7 QMCSampler

The best scores achieved by `QMCSampler`, both in terms of Accuracy and in terms of regularized score, are basically equivalent to the results of `RandomSampler`. In order to give more possibilities to the sampler, various initialization values were tried, none of which changed the results. It is not completely clear why this sampler produces results equivalent to those of `RandomSampler`, this will be further discussed in the Discussion section.

4.2.4 Discussion of the Results

This experiment, with the execution of these studies, allowed to establish important ground knowledge about the efficacy of the most important sampling techniques of Hyperparameter Optimization.

4.2.4.1 Analysis of the Best Hyperparameters Results

For the record, even if it is not in the main objectives of the experiment, the best hyperparameters found by the most reliable Sampler (`TPESampler`) were the following (Tab. 4.2.3). It is vital to note that, given the simplicity of the dataset, in some exe-

Table 4.2.3: *Best hyperparameters of Experiment 2, found by with the optimization process with* `TPESampler`.

Hyperparamter	Value
Activation	ReLU
Loss	CrossEntropy
Architecture	[input, 16, output]
Batch Size	16
Optimizer	Adam

cutions, the best hyperparameters' values were different (except model architecture), outlining how in simple problems like this one, the only really impactful factor was the architecture of the NN, while the possible combinations of the other hyperparameters able to achieve high results were many.

4.2.4.2 Analysis of the Failed Samplers

With regard to the two failed sub-experiments, which are, the studies using respectively `CmaEsSampler` and `PartiallyFixedSampler`, for the sake of simplicity, these samplers will be discarded. In order to conduct the next experiments, it was already necessary to identify the best samplers and discard the worst ones; so, discarding the failing samplers is the least time-consuming solution.

4.2.4.3 Analysis of the Worst Performing Samplers

As for the performance of `GPSampler`, `NSGAIISampler` and `QMCSampler`, one possible hypothesis for the reason their results are equivalent to `RandomSampler`, is the simplicity of the dataset and the model.

The hypothesis is, given that, as mentioned above, there is only one impactful hyperparameter, it is unlikely for these more complex sampling algorithms to find correlations in the Search Space, making the search partially unsuccessful. Therefore, for the next experiments, these three samplers will be discarded.

4.2.4.4 Analysis of the Best Samplers

The best results were obtained by the Study using `TPESampler`. The trend in the results suggests that the `TPESampler` is actually learning from past trials throughout the optimization process, as the best results are mostly in the later stages of the optimization.

Therefore, with regard to the first objective of the experiment, it can be concluded that `TPESampler` can be used for the other experiments. Together with `TPESampler`,

also `RandomSampler` can be used, because it did not show any unexpected behaviour and can thus be applied to make first assessments.

In conclusion, now that the best performing samplers are well-established, in the next experiments they can be utilized as a tool of comparison with other new sampling techniques.

4.2.4.5 Analysis of the Best NN Architectures

For what concerns the second objective of the experiment, which is to find lightweight, but well-performing nonetheless solutions, `TPESampler` gained interesting results.

The theoretical hypothesis was that, at the same total number of neurons, deeper architectures would have reached better scores. The results suggest, however, that there is no difference between the two, sometimes even linear architectures have achieved a good score.

From these results, it can be concluded that the experiment, and especially the model, are too simple for the score to be significantly different between deeper and less deep networks. Deeper networks do not result in an increase in generalization power if the model and the experiment are excessively simple.

4.3 Experiment 3 - Comparison between a Custom PSO implementation and Optuna

The goal of this experiment is to test and evaluate the functioning and performance of the custom implementation of Particle Swarm Optimization, adapted for HPO in a way that is similar to Optuna, and compare such performance to that of the samplers of Optuna.

In order to make the results and the methodology as much scientifically comparable as possible, as described in the methodology chapter (Sec. 3.2), the implementation of PSO replicates almost perfectly the functioning of Optuna and displays the result in the same exact way.

As the goal of the experiment is primarily to compare a new technique to existent techniques, and therefore would be excessive to operate on a complex and expensive problem, a simple dataset is considered, which again is MNIST.

System used for the Experiment:

The hardware that was utilized to run this experiment is the same as Experiment 2 (Sec. 4.2).

4.3.1 Preparation of the Experiment

The preparation for the experiment is similar to Experiment 2 (Sec. 4.2): the Studies related to the samplers of Optuna continue to be initialized with the "wrapping", whereas the Study for the custom PSO is initialized separately. Nevertheless, the process is basically the same, given that it is supposed to be a sort of replica of Optuna.

The MNIST dataset was imported through the `MNIST` package of Pytorch, included in torchvision, and then loaded with PyTorch data loaders.

4.3.1.1 Setup of the Studies

The Samplers of Optuna subject of this experiment are the following: `RandomSampler`, `TPESampler`. For each of the Samplers, an Optuna `Study` is created. All the samplers are initialized with their default input values. The Study of the custom PSO is called `PSO_optimization`.

As a result of the subsequent lack of scientific comparability between the two approaches if the pruning mechanism was involved for both, the Pruner is disabled.

4.3.1.2 Setup of the Objective Function

The objective function is the same for each Study, even for the `PSO_optimization` Study. The structure of the objective function is the same as the last experiment.

The Hyperparameters declared in the objective function, which form the Search Space of the Study, for this experiment were the following (Tab. 4.3.1).

Table 4.3.1: *Search Space of Experiment 3: list of hyperparameters and their values ranges, with the step. Where Step is $\sim$ if the range is continuous.*

Hyperparamter	Range	Step
Activation Function	{ReLU, tanh, sigmoid}	-
Hidden Layer Width 1	[0, 128]	1
Hidden Layer Width 2	[0, 128]	1
Hidden Layer Width 3	[0, 128]	1
Learning Rate	$[10^{-4}, 10^{-2}]$	$\sim$
Optimizer	{SGD, Adam}	-

Other parameters, related to the Optimization process, basically hyperparameters of the HPO process itself (could be called "Hyper-Hyperparamters"), which were used are the following (Tab. 4.3.2).

Table 4.3.2: *External optimization parameters of Experiment 3, basically hyperparameters of the optimization process, with their values.*

Group	Parameter	Value
Regularizer	labda	0.4
EarlyStopper	patience	5
PSOStopping	patience	5
	tolerance	0.005
PSO	num_particles	32
	max_generations	8

4.3.2 Execution of the Experiment

The execution of each Study is handled by an `OptunaRunner` object for Optuna and by `PSORunner` for PSO (it is exactly the same); which, at the end of the optimization process, saves the results on a csv file.

The number of Trials executed for each Study was 256.

In the reporting of the results, the information regarding what the best hyperparameters' values are will be ignored, because it is not the focus of the experiment, which is to identify how well performing the various techniques are.

4.3.2.1 RandomSampler

The best score achieved by the `RandomSampler` is 92%; there are 42 trials with a score over 80% and 2 with a score over 90%. The Accuracy of these top trials is around 95%-97%.

In terms of Accuracy, the best score is 98% and the best results are around 97%, but these results have far lower regularized scores.

As every experiment, these results from the `RandomSampler` have the purpose of evaluating how well the new techniques are performing, and whether they are doing what supposed to or are just collapsing into a random sampling.

4.3.2.2 TPESampler

The best score achieved by the `TPESampler` is 93.2%, there are 224 trials with a score over 80%; 135 with a score over 90%; 5 with a score over 93%. The Accuracy of these top trials is around 95%-96%.

In terms of Accuracy, the best scores are all around 97%, but these results have far lower regularized scores.

4.3.2.3 PSO_optimization

The best score achieved by the custom PSO algorithm is 93%, there are 123 trials with a score over 80%; 22 with a score over 90%; 2 with a score over 93%. The Accuracy of these top trials is around 95%-96%.

In terms of Accuracy, the best scores are all close to 98%, but these results have far lower regularized scores.

4.3.3 Discussion of the Results

The results of the experiment allow to safely conclude that the proposed adaptation of PSO to HPO was successful and is also perfectly comparable with the results from the Studies of Optuna.

4.3.3.1 Analysis on the Custom PSO Results

The results presented by the `PSO_optimization` Study, allow to establish that the algorithm is working as intended, the process is learning from past Trials. Analysing the results, it appears evident that the best scores have been obtained at the latest generations, outlining how the Particles are correctly closing their distance with the global optimum over time.

4.3.3.2 Analysis of the Comparison with Optuna

After comparing the results from the `PSO_optimization` Study, `RandomSampler` and `TPESampler`, it is easy to see that the PSO reaches similar levels of quality.

The comparison with `RandomSampler` clearly shows that the PSO is not sampling randomly, but is following a logic, based on the PSO algorithm.

The comparison with `TPESampler` allows to say that PSO is also extremely well-performing, and can be used in the same way as the TPESampler. The former has indeed found a greater number of very good solutions, but this is also because PSO has the tendency to never stop the exploration; in fact, watching the trial number around which the two algorithms started to find good solutions, it is almost the same; the main difference is a result of the characteristic of the TPE to focus on exploitation toward the last trials, which in this case led to slightly better results compared to PSO.

4.4 Experiment 4 - Comparison between a PSO Sampler and Optuna's Samplers

In the previous experiment, a custom implementation of PSO external to Optuna was tested, evaluated, and compared to the performance of Optuna.

This experiment instead, aims to test, evaluate, and compare the `PSOSampler`, a Sampler developed within Optuna that implements the PSO algorithm, described in the Methodology chapter in the related section (Sec. 3.3.2).

In this way, the results will already be perfectly comparable to the ones of the native samplers of Optuna. The same exact code infrastructure utilized for the Studies of Optuna, can also be used for the Study which uses the `PSOSampler`.

The experiment is divided into two sub-experiments, with the same approach and the same goals. The only difference between the two is that in the first Pruning is disable, in the second is enabled.

As the goal of the experiment is primarily to compare a new technique to existent techniques, and therefore would be excessive to operate on a complex and expensive problem, a simple dataset is considered, which again is MNIST.

System used for the Experiment:

The hardware that was utilized to run this experiment is the same as Experiment 2 (Sec. 4.2).

4.4.1 Preparation of the Experiment

The preparation for the experiment is very similar to Experiment 2 (Sec. 4.2): now `PSOSampler` has become a Sampler of Optuna, thus initializing its related `Study` object is the same procedure as the standard Studies of Optuna; they continue to be initialized with the "wrapping".

The MNIST dataset was imported through the `MNIST` package of Pytorch, included in torchvision, and then loaded with PyTorch data loaders.

4.4.1.1 Setup of the Studies

The samplers of Optuna subject of this experiment are the following: `RandomSampler`, `TPESampler`, `PSOSampler`. For each of the samplers, an Optuna `Study` is created. All the standard samplers are initialized with their default input values. The `PSOSampler` is initialized with two values, `num_particles` and `max_generations`.

For the experiment where Pruning is enabled, the Hyperband algorithm is used, applied through the `HyperbandPruner` object of Optuna.

4.4.1.2 Setup of the Objective Function

The objective function, as always, is the same for each Study. The structure of the objective function is the same as in the last experiments.

The Hyperparameters declared in the objective function, which form the Search Space of the Study, for this experiment were the following (Tab. 4.4.1).

Table 4.4.1: *Search Space of Experiment 4: list of hyperparameters and their values ranges, with the step. Where Step is ∼ if the range is continuous.*

Hyperparamter	**Range**	**Step**
Activation Function	{ReLU, tanh, sigmoid}	-
Hidden Layer Width 1	[0, 128]	8
Hidden Layer Width 2	[0, 128]	8
Hidden Layer Width 3	[0, 128]	8
Learning Rate	$[10^{-4}, 10^{-2}]$	∼
Optimizer	{SGD, Adam}	-

Other parameters, related to the Optimization process, basically hyperparameters of the HPO process itself (could be called "Hyper-Hyperparamters"), which were used are the following (Tab. 4.4.2).

Table 4.4.2: *External optimization parameters of Experiment 4, basically hyperparameters of the optimization process, with their values.*

Group	**Parameter**	**Value**
Regularizer	labda	0.4
EarlyStopper	patience	5
PSOSampler	num_particles	32
	max_generations	8
HyperbandPruner	min_resources	3
	max_resources	30
	reduction_factor	3
	bootstrap_count	6

4.4.2 Execution of the Experiment

The execution of each Study is handled by an `OptunaRunner` object for Optuna, which at the end of the optimization process saves the results on a csv file.

The number of Trials executed for each Study was 256.

In the reporting of the results, the information regarding what the best hyperparameters' values are will be ignored, because it is not the focus of the experiment, which is to identify how well performing the various techniques are.

4.4.2.1 RandomSampler vs TPESampler vs PSOSampler - No Pruning

The best score achieved by the `RandomSampler` is 92%; there are 42 trials with a score over 80% and 2 with a score over 90%. The Accuracy of these top trials is around 95%-97%.

The best score achieved by the `TPESampler` is 93.2%; there are 224 trials with a score over 80%; 135 with a score over 90%; 5 with a score over 93%. The Accuracy of these top trials is around 95%-96%.

The best score achieved by the `PSOSampler` is 93.8%; there are 174 trials with a score over 80% and 117 with a score over 90%. 28 with a score over 93%. The Accuracy of these top trials is around 95%-96%.

4.4.2.2 RandomSampler vs TPESampler vs PSOSampler - With Pruning

The best score achieved by the `RandomSampler` is 90.6%; there are 29 trials with a score over 80% and 1 with a score over 90%. The Accuracy of these top trials is around 91%-97%.

The best score achieved by the `TPESampler` is 94%; there are 216 trials with a score over 80%; 105 with a score over 90%; 41 with a score over 93%. The Accuracy of these top trials is around 95%-96.5%.

The best score achieved by the `PSOSampler` is 93.4%; there are 156 trials with a score over 80% and 89 with a score over 90%. 9 with a score over 93%. The Accuracy of these top trials is around 94%-95%.

4.4.3 Discussion of the Results

The results of the experiment, in the first place, demonstrated the validity of the `PSOSampler`, showing that it works as supposed and is perfectly integrated in Optuna, being executed without errors and reaching good results. Another really positive outcome, is the confirmation of the applicability of PSO to Hyperparameter Optimization. This experiment further proved what was already being suggested by the previous one.

4.4.3.1 Analysis of the Results of PSOSampler

The `PSOSampler` is clearly working as intended, and the implementation of a custom sampler within Optuna has been revealed to be successful. The `PSOSampler`, as

confirmed by the results, obtained top-level results; finding the best among possible candidate solutions for the given problem.

4.4.3.2 Analysis of the Comparison with Optuna

The results showed that `PSOSampler` can even outperform the most powerful Optuna's samplers, in fact, the `PSOSampler` had better results than the `TPESampler` in the sub-experiment without pruning. In the other sub-experiment, the one with pruning, it reached slightly worse results than the `TPESampler`. Intuitively, it makes sense for a PSO algorithm to suffer pruning, because in order for its functioning to be as much accurate as possible, it needs all the information regarding the Search Space, and the concept of pruning remove some of that information.

4.4.3.3 Analysis of the Trials Performance Distribution

In order to show how the Samplers improve over time, a scatter plot can be used, with the Trial number on one axis and the score on the other.

In the `RandomSampler` is evident how the search is searching randomly within the space, there is no learning from the past trials (Fig. 4.4.1).

In the `TPESampler` it can be observed how, even after relatively few trials, the score starts becoming better and better, learning from previous trials (Fig. 4.4.2).

In the `PSOSampler`, as in the `TPESampler`, the algorithm learns from the past trials. It can be seen how this "convergence" happens later compared to the `TPESampler`, this is because PSO is initially an explorative technique, that only in later generations starts to accumulate trials toward the best solution (Fig. 4.4.3).

4.4.3.4 Analysis of the Hyperparameters Importance

In this experiment, being all the sub-experiments part of Optuna, it was possible to use Optuna Storage to generate some plots.

All three Samplers generated similar Hyperparameters Importance histograms. In all three, it is evident how the three hyperparameters associated with the layer width are not at the same level of importance, when they should be (Fig. 4.4.4). The reason for this is not entirely clear, but it is probably again due to the fact that the problem is very simple, so almost any change in the hyperparameters can lead to a good result, moving away the focus from the importance of the architecture complexity.

The most important hyperparameter appears to be the optimizer; this is probably as a result of the fact that there are only two optimizers, and Adam is almost always better than SGD, therefore making the final score obtain a great advantage.

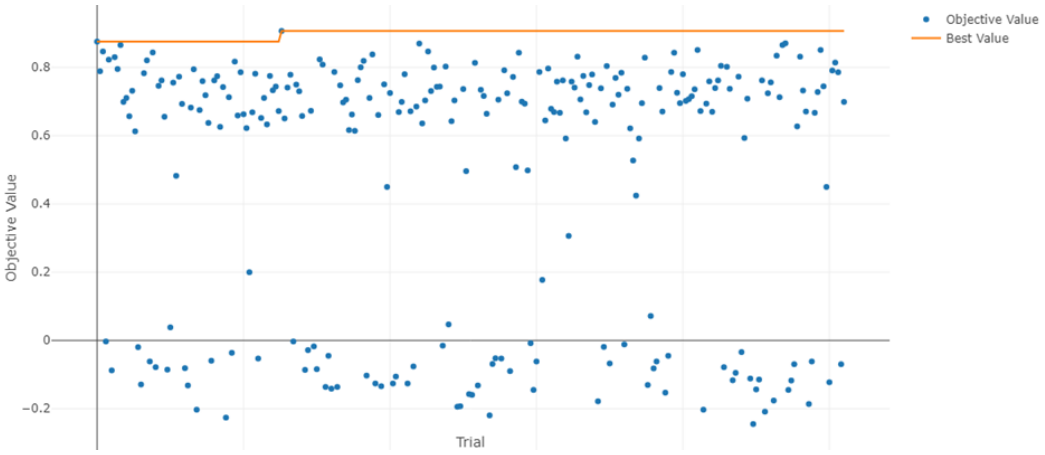

Figure 4.4.1: *Scatter plot of the distribution of the performance of the trials of the RandomSampler in Experiment 4.*

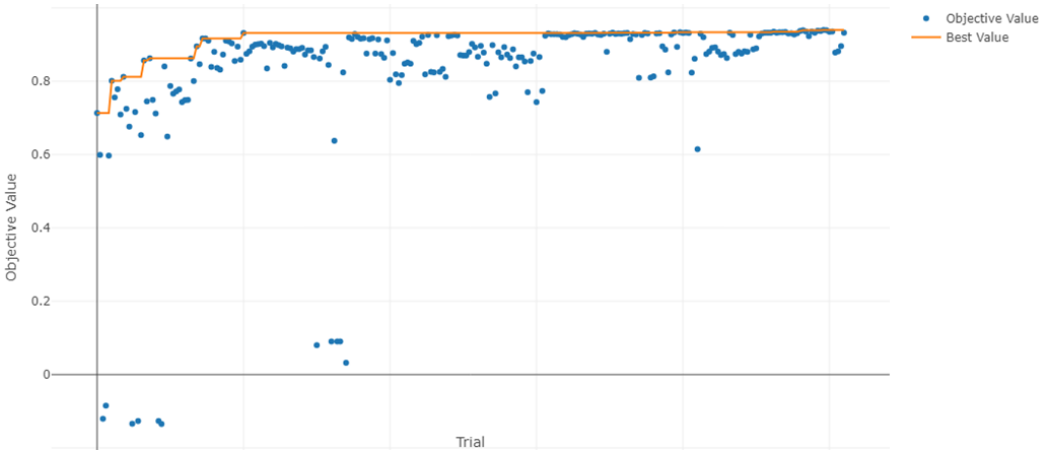

Figure 4.4.2: *Scatter plot of the distribution of the performance of the trials of the TPESampler in Experiment 4.*

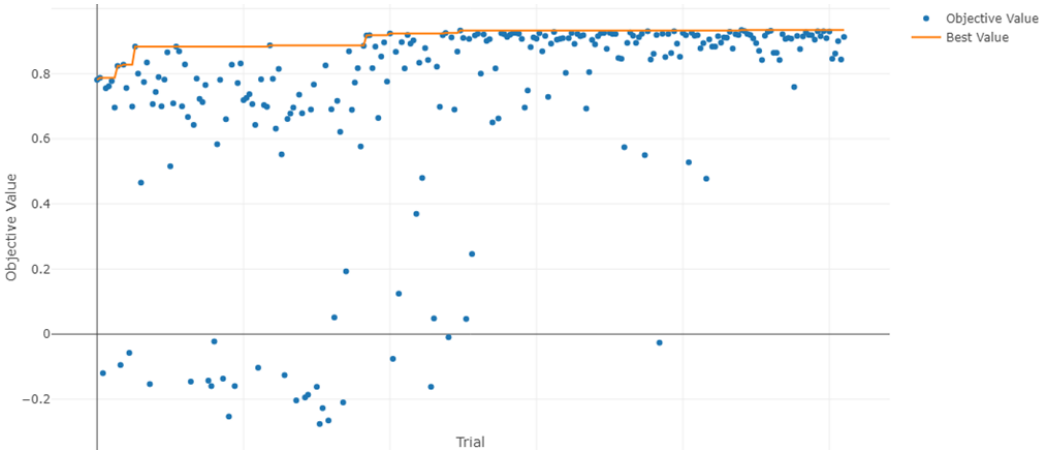

Figure 4.4.3: *Scatter plot of the distribution of the performance of the trials of the PSOSampler in Experiment 4.*

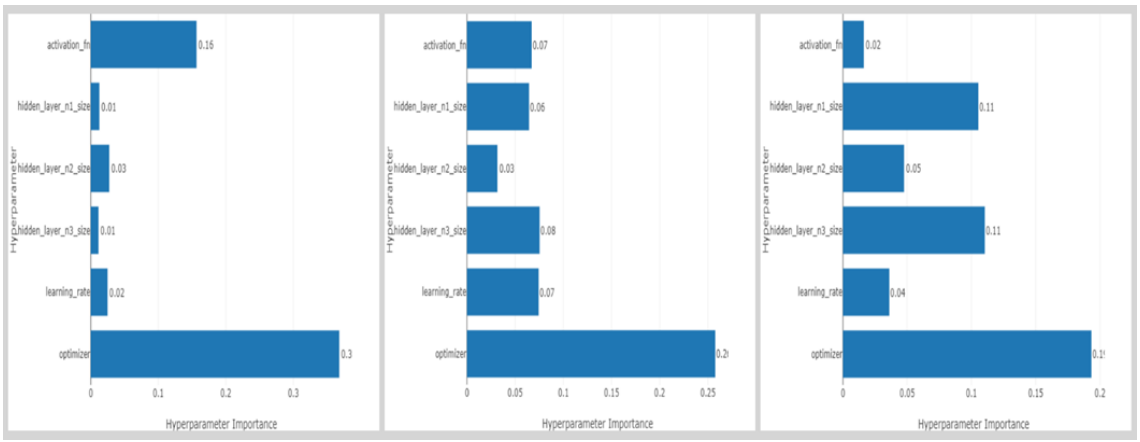

Figure 4.4.4: *Comparison between the generated histograms of the importance of the hyperparameters of Experiment 4. From left to right, Studies of: RandomSampler, TPESampler, PSOSampler.*

4.5 Experiment 5 - Hyperparameter Optimization on Weed Map Dataset

The goal of this experiment is to perform a Hyperparameter Optimization, using the samplers from the previous experiments, for a Weed Mapping problem.

In Drone Vision, Semantic Segmentation is applied to identify immediately and automatically the different categories of "object" inside a cultivated field (crop, terrain, weed). But drones, especially those used for agriculture, do not carry powerful computers able to run complex Neural Networks Models.

Therefore, it is necessary to execute a Hyperparameter Optimization of a NN model for Drone Vision, in order to find a candidate solution that is lightweight but does not lose too much generalization power.

The model utilized for this experiment is Lawin, which is introduced in its relative section in the Methodology chapter (Sec. 3.4.1.2).

The Dataset of the experiment is Weed Map, a popular dataset for Drone Vision.

System used for the Experiment:

The hardware that was utilized to run this experiment is the following:

- CPU: AMD EPYC 7V12(Rome) @ 2.45GHz (4-core)
- GPU: NVIDIA Tesla T4 (16GB VRAM)
- RAM: 28GB

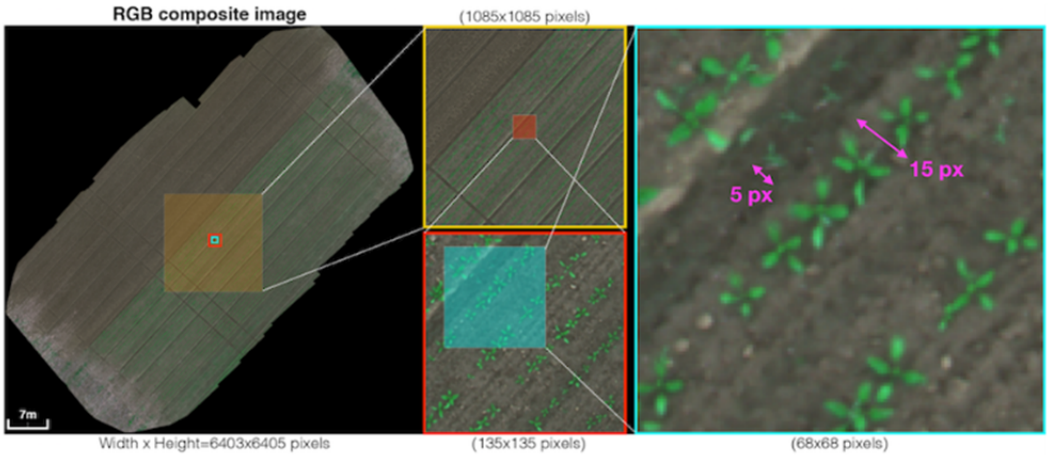

Figure 4.5.1: *Example of image in Weed Map Dataset; with different zoom levels. Source: [34]*

- Disk: SSD
- OS: Linux (Ubuntu)

(This hardware is a Virtual Machine on the Azure ML servers).

4.5.1 Weed Map Dataset

4.5.1.1 Introduction to Weed Map Dataset

The Weed Map Dataset is the dataset for the research "WeedMap: A large-scale semantic weed mapping framework using aerial multispectral imaging and deep neural network for precision farming" [34].

Each picture of the Dataset shows an image of a cultivated field, where there can be distinguished Dirt, Weeds and Crops.

The image below (Fig. 4.5.1) is an example of one of the images. It is first shown the entire Orthomosaic Map, and then different zoom levels. This highlights the large scale of the image and the difficulty in distinguishing between Crops and Weeds.

Data Collection:

The data included in the dataset was collected from aerial images (with Drones) from two different Sugar Beet fields: Eschikon (Switzerland) and Rheinbach (Germany). (Fig. 4.5.2)

In particular, the utilized Drones are two commercial quadrotor UAVs, equipped with multispectral cameras.

Description	1st campaign	2nd campaign
Location	Eschikon, Switzerland	Rheinbach, Germany
Date, Time	5-18/May/2017, around 12:00 PM	18/Sep/2017, 9:18-40 AM
Aerial platform	Mavic pro	Inspire 2
Sensor[a]	Sequoia	RedEdge
# Orthomosaic map	3	5
Training/Testing multispectral images[b]	227/210	403/94
Crop	Sugar beet	
Altitude	10 m	

Figure 4.5.2: *Information on the campaigns of data collection of Weed Map Dataset. Source: [34]*

4.5.1.2 Structure of Weed Map Dataset

The full Dataset consist of 129 directories, for a total of 18746 image files. The total weight is 5.36GB.

Folder Structure:

There are two main folders, which are Orthomosaic and Tiles. Orthomosaic contains the full orthomosaic maps. Tiles contains, for each orthomosaic map, a folder containing images that represent cropped sections of the original orthomosaic map. All cropped sections together form the full map.

Both Orthomosaic and Tiles contain RedEdge and Sequoia subfolders, containing 8 Orthomosaic maps in total (5 RedEdge and 3 Sequoia). (Fig. 4.5.3)

Each Orthomosaic map stands in a folder indexed from 000 to 007.

Groundtruth:

The Groundtruth images are pictures of the fields where each pixel has been manually labeled. (Fig. 4.5.4) Each class is represented in a different colour:

- Background (Dirt and part and part of the image which is not the crop field): is Black (code 0)
- Crops: is Green (code 1)
- Weeds: is Red (code 2)

The Groundtruth images are utilized to evaluate the precision of each prediction.

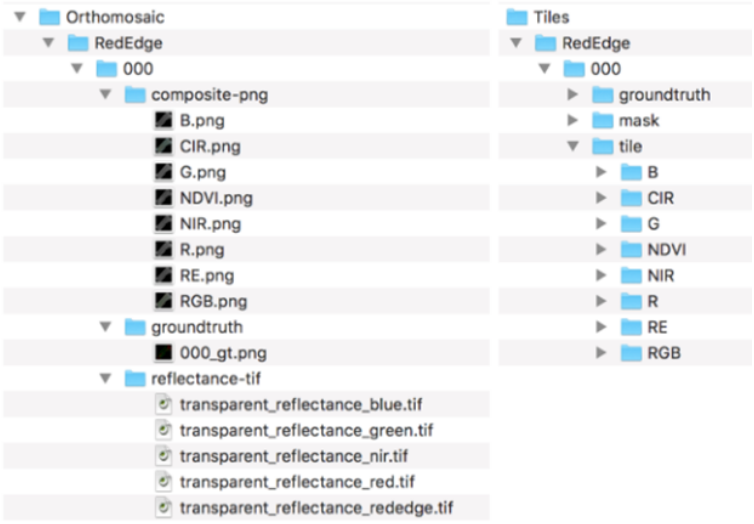

Figure 4.5.3: *Table showing the folder structure of Weed Map Dataset. Source: [34]*

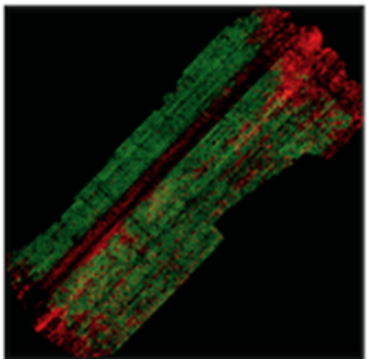

Figure 4.5.4: *Example of GroundTruth image in Weed Map Dataset. Source: [34]*

4.5.2 Preparation of the Experiment

This experiment is partially different from the previous experiments. All studies continue to be initialized with the "wrapping".

The architectures with which to initialize the NN are no longer built inside the objective function, and thus need to be defined earlier. The architectures subject of this experiment are defined in the same article in which the original experiment of Weed Mapping was proposed [30]. During the experiment, those architectures will be referenced as "backbone".

The Weed Map dataset was loaded using the pre-defined function mentioned in the related section of the Methodology chapter (Sec. 3.4.1.2). The utilized version of Weed Map Dataset is not the original version, but is a pre-processed one (the same as [30]).

4.5.2.1 Setup of the Studies

The samplers of Optuna used to perform three different HPOs, are the following: `RandomSampler`, `TPESampler`, `PSOSampler`. For each of the samplers, an Optuna `Study` is created. All the standard samplers are initialized with their default input values. The `PSOSampler` is initialized with two values, `num_particles` and `max_generations`.

The Pruning mechanism used is the Median Pruning algorithm. It is applied through the `MedianPruner` object of Optuna, whose values for initialization are below.

4.5.2.2 Setup of the Objective Function

The objective function, as always, is the same for each Study. The structure of the objective function is the same as in the last experiments.

It is important to note one difference, which is the metric to follow. Although all four metrics are always computed, in this experiment the F1-Score will be the main metric, differently from the previous experiments where it was Accuracy.

The initialization hyperparameters of the loss function, `Focal`, were fixed. Their chosen values are: `gamma` = 2, `weights` = [0.06, 1, 1.7].

The Hyperparameters declared in the objective function, which form the Search Space of the Study, for this experiment were the following (Tab. 4.5.1).

Table 4.5.1: *Search Space of Experiment 5: list of hyperparameters and their values ranges, with the step. Where Step is $\sim$ if the range is continuous.*

Hyperparamter	Range	Step
Backbone	{MiT-B0, MiT-LD, MiT-L0, MiT-L1, MiT-L2}	-
Learning Rate	$[10^{-4}, 10^{-2}]$	$\sim$
Optimizer	{SGD, Adam}	-

Other parameters, related to the Optimization process, basically hyperparameters of the HPO process itself (could be called "Hyper-Hyperparamters"), which were used are the following (Tab. 4.5.2).

4.5.3 Execution of the Experiment

The execution of each Study is handled by an OptunaRunner object for Optuna, which at the end of the optimization process saves the results on a csv file.

The number of Trials executed for each Study was 64.

Table 4.5.2: *External optimization parameters of Experiment 5, basically hyperparameters of the optimization process, with their values.*

Group	Parameter	Value
Regularizer	labda	0.4
EarlyStopper	patience	15
PSOSampler	num_particles	32
	max_generations	8
MedianPruner	n_startup_trials	2
	n_warmup_steps	20
	interval_steps	20
	n_min_trials	4

4.5.3.1 Score Results

The best score achieved by the `RandomSampler` is 70%; the top trials have as backbone MiT-L1 and MiT-L2; F1-Score of the best trials is around 69%-74%.

The best score achieved by the `TPESampler` is 74.4%; the top trials have as backbone MiT-L2; the F1-Score of the best trials is around 70%-77%.

The best score achieved by the `PSOSampler` is 70.1%; the top trials have as backbone MiT-L2 (the best one MiT-L1); the F1-Score of the best trials is around 70%-75%.

For all three samplers, the best trials in terms of F1-Score have as backbones MiT-B0 and MiT-LD.

4.5.3.2 Hyperparameters Results

The Hyperparameters Importance graph is shown in the figure (Fig. 4.5.5), which is the one generated from `TPESampler`, being the Study with the best results, even though also the other two studies generated very similar results.

The best candidate solution, and thus the best found hyperparameters are:

- Backbone = MiT-L2
- Learning Rate = 0.002
- Optimizer = Adam

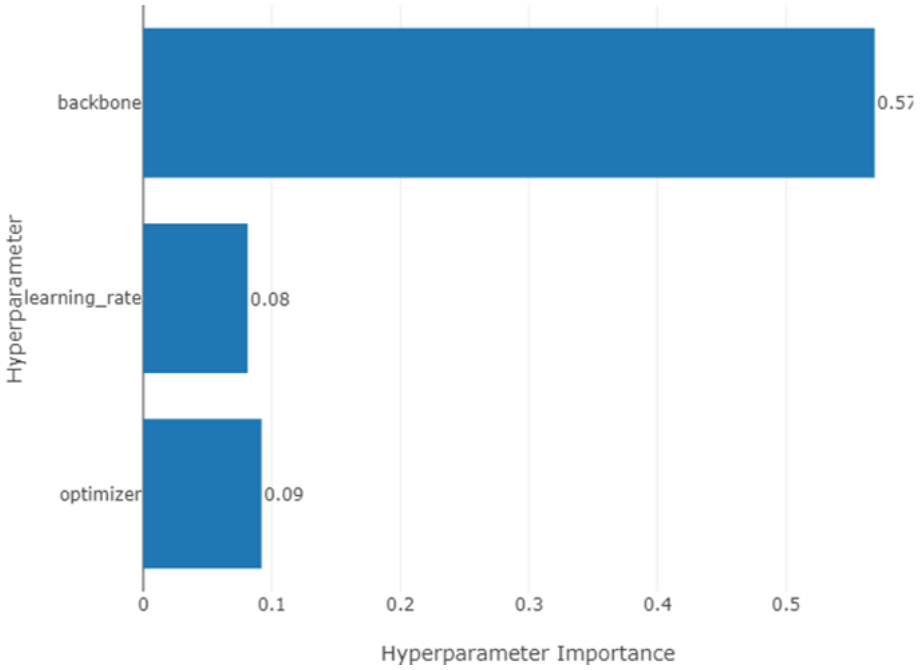

Figure 4.5.5: *Histogram of the importance of the hyperparameters of Experiment 5, generated by the Study of TPESampler.*

4.5.4 Discussion of the Results

The results the of experiment allowed to find a valid hyperparameters' configuration that solves the initial problem. In fact, the proposed solution, uses the most lightweight among all the Backbones, creating the most lightweight Neural Network for the Drone Vision problem possible.

4.5.4.1 Analysis of the Performance of the Samplers

The `RandomSampler` worked as usual, there does not seem to be any problem with its functioning. The `TPESampler` found good solutions, therefore, it could be argued that it is working as supposed. The `PSOSampler` on the other hand, showed results slightly below expectations.

The reason `PSOSampler` underperformed in this particular experiment, is because of the way pruning influences the sampler in more complex problems like this experiment. Differently from `TPESampler`, where there is only a "global perspective", `PSOSampler` suffers a significantly more from pruning. This is because the PSO algorithm is based on the concept of particles that move in a space, and the pruning mechanism removes some of the information that the particles need to move correctly. Specifically, the trials of the particles that are not near the global optimum are pruned most of the time, not allowing the particles to update their Local Optimum. Thus, the Cognitive Factor of those particles becomes basically useless, and only the Social Factor matters.

This causes a slowdown in the convergence of the algorithm, leading to worse results compared to other samplers like `TPESampler`.

Potential fixes for this problem could be to remove the pruning mechanism for the `PSOSampler` (but this would make the process very computationally expensive), or to modify the pruning mechanism to be less aggressive, to allow the worst-positioned particles to continue to explore the space.

4.5.4.2 Analysis of the Quality of Results

The considered results for this analysis are the ones from the `TPESampler`, as it was the sub-experiment with the best results among all samplers.

The best solutions all have in common the same Backbone, which is MiT-L2. This is the lightest Backbone among the proposed ones. Tested singularly with already known good hyperparameters, the MiT-B0 and MiT-LD, which are deeper, more complex architectures, have obtained results around 76% of F1-Score on the Test Set. The best solution obtained by the `TPESampler` with the MiT-L2 architecture obtained a F1-Score of 74.4%, which is only 1.6% apart from the best architecture.

Therefore, this experiment could be considered a success, as the architecture of the Neural Network was heavily simplified from a total of 1024 neurons to 64 (MiT-B0 and MiT-L2 respectively), losing only 1.6% from the performance.

Chapter 5

Conclusions

The results of the experiments allowed to reach significant conclusions about Hyperparamter Optimization and its related techniques. The work was able to outline the importance of HPO and evaluate its effectiveness in various contexts with the use of popular and new techniques.

5.1 Summary of the Results

In Experiment 1, it was possible to demonstrate that HPO is a powerful strategy to improve the performance of machine learning models; improvements in performance were clearly observable in the results of the experiments, showing that emphirical or random approaches to the Tuning of Hyperparameters are not as effective as a formal optimization process such as HPO.

In Experiment 2, the applicability of a HPO's library, Optuna, was tested and evaluated. In particular, different HPO algorithms (those with an implementation in Optuna) were tested and compared on the classic MNIST dataset. The results showed that the TPE sampler was the most effective in this case, but it is vital to note that the performance of the algorithms may vary depending on the dataset and the model being used. It was also observed that a simple method, such as Random Seach, can be utilized as a baseline for comparison with other HPO algorithms.

In Experiment 3, using an approach basically identical to Optuna's, a framework for HPO's was built, that used Particle Swarm Optimization (PSO) as the sampling algorithm. The results showed that the PSO algorithm was not as effective as the TPE sampler from Optuna, but it was still able to improve the performance of the models, reaching similar levels of quality performance-wise. The experiment also demonstrated that building a framework for HPO is a viable strategy, and can save time and effort in the preparation phase of the Hyperparameters Tuning process.

In Experiment 4, a sampler using the PSO algorithm was implemented in Optuna. Firstly, the implementation of the sampler was revealed as a success, integrating the

PSO algorithm into the Optuna library. Secondly, the results of the experiments showed that the created PSO sampler reached similar performance levels to the TPE sampler, even surpassing it in some cases. This further demonstrated the effectiveness of the PSO algorithm as a sampling method for HPO.

In Experiment 5, the strategies utilized so far were tested on a more complex dataset, the Weed Map dataset. The results allowed in the first place not only to find the best hyperparameters for the WeedMap problem, but also to simplify the architecture of the Nueral Network used in the experiment, with a relatively small loss in the performance of the model. The results also showed, not without struggles, that the HPO algorithms were also able to work effectively in a more complex situation, in particular, PSO can now be considered a valid alternative to the TPE sampler in the Optuna library, and, in general, a top-level sampling technique in HPO.

5.2 Limitations

There are some limitations in the presented results that need to be considered. These limitations are mainly related to the intrinsic complexity of HPO from a computational cost point of view.

In general, all strategies revealed themselves effective only when a large number of trials were performed, and obviously, the more trials are performed, the more time is needed, and the more time is needed, the more computational resources are required to make the process feasible. This limitation is particularly evident when using the PSO sampler, which showed, most of the time, to require more trials than the TPE sampler to reach a satisfactory level of performance. The limitation did not impact the results of the first four experiments, as both the model and the dataset were simple, therefore requiring less time to complete a single trial, and fewer total trials. The problems, however, became evident in the last experiment, where the complexity of the dataset and the model required a large number of trials to reach a good level of performance. Even with relatively good, but not excellent, hardware, the time required to complete the trials was significantly higher than in the previous experiments, and this led to the necessity of reducing the number of trials to perform, so as to make the process feasible in a reasonable amount of time (below weeks).

Another limitation to be considered, is the lack of a more detailed analysis of other HPO algorithms, starting with the ones implemented in Optuna, which were given less importance in the experiments. As for the objectives of the experiments, it was not really necessary to test all the algorithms available in Optuna deeply, therefore, when one technique showed problems, for the sake of simplicity, it was decided to ignore it and move on. This limitation actually reduces the generalizability of the results, especially for what concerns Experiment 2.

5.3 Future Work

Potential future work should focus on addressing the limitations of the current work and further exploring the potential of HPO and its related techniques.

A first recommendation is to reproduce Experiments 2 and 4 on a more complex dataset, additionally, more HPO algorithms should be tested and compared. Those might be: samplers already implemented in Optuna; existing algorithms to adapt for HPO, implementing them from scratch; creating brand new HPO algorithms.

A second recommendation is to execute Experiment 5 on top-level hardware, in order to execute the optimization with more trials. This would allow to reach a better level of performance, and better compare the PSO sampler with the TPE sampler.

A third recommendation concerns the PSO algorithm for HPO. The results of Experiment 5 showed how the PSOSampler suffers from Pruning more than other samplers. Future work should focus on developing specialized pruning techniques for the PSO algorithm, for instance, a pruning technique less aggressive toward particles far from the Global Optimum. Alternatively, a Pruning technique that executes a single pruning process for each particle individually.

The results of Experiment 4 are a fundamental baseline for a potential pull-request to the Optuna library, in order to integrate the PSO sampler into the official release of the library. This would make the PSO sampler available to a wider audience. Before doing this, however, it is recommended to further test the sampler on different datasets and models, in order to ensure its effectiveness in different contexts.

Hyperparamter Optimization is a sub-field of Machine Learning that is still in its infancy. Its main limitation lies in the computational cost required to perform the optimization. In the future, hardware might be able to overcome this limitation, by then, research in HPO should focus on the development of more efficient algorithms, adapting already existing optimization techniques, or creating new ones specifically designed for HPO.

Bibliography

[1] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2020.07.061. URL https://www.sciencedirect.com/science/article/pii/S0925231220311693.

[2] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. URL https://d2l.ai/chapter_hyperparameter-optimization/index.html.

[3] Wikipedia contributors. Hyperparameter optimization — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Hyperparameter_optimization&oldid=1193594191. [Online; accessed 14-May-2024].

[4] Antonin Raffin. Automatic Hyperparameter Tuning - A Visual Guide, 2023. URL https://araffin.github.io/post/hyperparam-tuning/.

[5] Radwa Elshawi, Mohamed Maher, and Sherif Sakr. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287*, 2019.

[6] Sayak Paul. Hyperparameter Optimization in Machine Learning Models, 2021. URL https://www.datacamp.com/tutorial/parameter-optimization-machine-learning-models.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(null):281–305, feb 2012. ISSN 1532-4435.

[8] Jason Brownlee. Hyperparameter Optimization With Random Search and Grid Search, 2020. URL https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/.

[9] Shuhei Watanabe. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *arXiv preprint arXiv:2304.11127*, 2023.

[10] Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, Difan Deng, and Marius Lindauer. Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *WIREs Data Mining and Knowledge Discovery*, 13(2):e1484, 2023. doi: https://doi.org/10.1002/widm.1484. URL `https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1484`.

[11] Jonathan Lorraine and David Duvenaud. Stochastic hyperparameter optimization through hypernetworks. *arXiv preprint arXiv:1802.09419*, 2018.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6 (2):182–197, 2002. doi: 10.1109/4235.996017.

[13] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. pages 312–317, 1996. doi: 10.1109/ICEC.1996.542381.

[14] scikit-learn developers. Tuning the hyper-parameters of an estimator, 2020. URL `https://scikit-learn.org/stable/modules/grid_search.html`.

[15] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1238–1246, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL `https://proceedings.mlr.press/v28/karnin13.html`.

[16] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL `http://jmlr.org/papers/v18/16-558.html`.

[17] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Bentzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 230–246, 2020. URL `https://proceedings.mlsys.org/paper_files/paper/2020/file/a06f20b349c6cf09a6b171c71b88bbfc-Paper.pdf`.

[18] Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*, 2020.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[20] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[21] datagy.io. Python Optuna: A Guide to Hyperparameter Optimization, 2023. URL `https://datagy.io/python-optuna/`.

[22] Filippo Chinni Carella. Bachelor-thesis, 2024. URL `https://github.com/FilippoChinniUNIBA/Bachelor-Thesis`.

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

[24] Adrian Tam. A Gentle Introduction to Particle Swarm Optimization, 2021. URL `https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/`.

[25] Axel Thevenot. Particle Swarm Optimization (PSO) Visually Explained, 2020. URL `https://towardsdatascience.com/particle-swarm-optimization-visually-explained-46289eeb2e14`.

[26] Dongshu Wang, Dapei Tan, and Lei Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22, 01 2018. doi: 10.1007/s00500-016-2474-6.

[27] Mark M Millonas. Swarms, phase transitions, and collective intelligence. *arXiv preprint adap-org/9306002*, 1993.

[28] Eberhart and Yuhui Shi. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 1, pages 81–86 vol. 1, 2001. doi: 10.1109/CEC.2001.934374.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[30] Giovanna Castellano, Pasquale De Marinis, and Gennaro Vessio. Weed mapping in multispectral drone imagery using lightweight vision transformers. *Neurocomputing*, 562:126914, 2023. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2023.126914. URL `https://www.sciencedirect.com/science/article/pii/S0925231223010378`.

[31] Pasquale De Marinis. Lwvits-for-weedmapping, 2023. URL `https://github.com/pasqualedem/LWViTs-for-weedmapping`.

[32] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[33] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[34] I. Sa, M. Popovic, R. Khanna, Z. Chen, P. Lottes, F. Liebisch, J. Nieto, C. Stachniss, A. Walter, and R. Siegwart. Weedmap: A large-scale semantic weed mapping framework using aerial multispectral imaging and deep neural network for precision farming. *MDPI Remote Sensing*, 10(9), Aug 2018. doi: doi:10.3390/rs10091423.