TÉCNICO LISBOA

Optimization of Machine Learning Jobs in the Cloud

João Pedro Neves Nogueira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Paolo Romano

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. João Coelho Garcia

January 2021

Acknowledgments

Foremost, would like to thank my Supervisor Professor Paolo Romano that has always guided me in the making of this thesis and also made the previously unfamiliar topic of Machine Learning optimization so interesting and fun to learn. I would also like to thank both Maria Casimiro and Pedro Mendes that were always there for support and provided valuable feedback and ideas to achieve our proposed system. Finally I would like to thank my family and Carolina for supporting me after long days and nights of work.

Resumo

Os sistemas de aprendizagem automática e de computação em nuvem tem sido duas das áreas de crescimento mais rápido nos últimos anos. Recentemente têm surgido inovações na area de arendizagem automática, melhorando a sua capacidade de atingir elevados níveis de precisão e reduzindo o seu tempo de treino. Para a otimização de sistemas de aprendizagem automática com datasets muito grandes e uma larga quantidade de hiper-parâmetros, a maioria de utilizadores recorre à computação em nuvem para conseguir proceder à sua otimização, facilitando a sua acessibilidade. Para fazer isto, os utilizadores enfrentam a barreira de escolher quais são os parâmetros de configuração na nuvem, para conseguirem executar os seus algoritmos de aprendizagem automática, o que pode ser bastante difícil devido à enorme escolha de configurações possíveis, dado que uma escolha errada destes parâmetros irá traduzir-se em custos elevados para modelos grandes. Recente sistemas de estado da arte têm optado por uma aproximação de otimização de ambos parâmetros de sistemas de aprendizagem automática e parâmetros de configuração de computação em nuvem, com o objetivo de minimizar custos relacionados com computação em nuvem enquanto se obtêm a melhor configuração de hiper-parâmetros para o específico algoritmo de aprendizagem automática. De qualquer modo, este processo de otimização também requer a exploração de muitas configurações e impõe grandes gastos económicos. Por esta razão é crucial que o algoritmo de otimização seja eficiente a nível de tempo, convergindo para a configuração ótima da forma mais rápida possivél. Esta dissertação propõe Hydra, um sistema que procede à otimização de sistemas de aprendizagem automática, melhorando desvantagens de sistemas de estado-da-arte que estende ao rapidamente convergir em direção à solução ótima sem perder tempo em treinar o modelo, usando muitas configurações de baixo custo enquanto aplica técnicas de transferência de conhecimento para melhorar o desempenho do modelo e, ultimamente a reduzir o custo final por 35%.

Palavras-chave: Sistemas de Aprendizagem Automática, Computação em Nuvem.

Para a otimização de sistemas de aprendizagem automática com datasets muito grandes e uma larga quantidade de hiper-parâmetros, a maioria de utilizadores recorre à computação em nuvem para conseguir proceder à sua otimização, facilitando a sua acessibilidade. Para fazer isto, os utilizadores enfrentam a barreira de escolher quais são os parâmetros de configuração na nuvem, para conseguirem executar os seus algoritmos de aprendizagem automática, o que pode ser bastante difícil devido à enorme escolha de configurações possíveis, dado que uma escolha errada destes parâmetros irá traduzir-se em custos elevados. Recentes sistemas de estado da arte têm optado por uma aproximação de otimização de ambos parâmetros de aprendizagem automática e de computação em nuvem, com o objetivo de minimizar custos relacionados enquanto se obtêm a melhor configuração de hiper-parâmetros para o específico algoritmo. Este processo de otimização também requer a exploração de muitas configurações e impõe grandes gastos económicos. Por esta razão é crucial que o algoritmo de otimização seja eficiente a nível de tempo, convergindo para a configuração ótima da forma mais rápida possivél. Esta dissertação propõe Hydra, um sistema que procede à otimização de sistemas de aprendizagem automática, melhorando desvantagens dos sistemas estado-da-arte integrados ao

rapidamente convergir em direção à solução ótima sem perder tempo em treinar o modelo, usando muitas configurações de baixo custo enquanto aplica técnicas de transferência de conhecimento para melhorar o desempenho do modelo e, ultimamente a reduzir o custo final por 35

Abstract

Machine Learning and Cloud Computing have been two of the fastest growing areas in the the past few years. Recent developments have emerged regarding machine learning optimizations, enhancing their accuracy and training time. However, for optimization procedures that have very large datasets and many hyperparameters, most users turn to the cloud to offload the inherent computation that would otherwise be infeasible locally. In order to do so, users face the task of picking cloud parameters to deploy their machine learning jobs which can be difficult due to the wide range of possible configurations and whose misconfiguration translates into large, unnecessary costs for large scale models. Recent state-of-the-art systems have taken the approach of performing optimization of both cloud configurations and machine learning hyperparameters in a joint fashion, with the goal of minimizing cloud related expenses while reaching the best hyperparameter configuration for the specified machine learning algorithm. Nonetheless, the optimization procedure involved by these approaches can also require exploring a large number of expensive configurations and impose, in its turn, large economical costs. It is thus crucial that the optimization procedure is as time efficient as possible and converges rapidly towards the optima. This thesis proposes Hydra, a self-tuning system solution that performs optimization of machine learning algorithms improving some drawbacks of extended state-of-the-art systems by rapidly converging towards the optimum solution without wasting time on bootstrapping the model, using many low-budget evaluations of configurations while applying transfer-learning to enhance the models' performance, ultimately reducing overall costs by 35% of the extended work.

Keywords: Machine Learning, Cloud Computing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Machine learning (ML) has emerged as a popular research area that aims at automating model building in order to develop self learning systems. ML branches from artificial intelligence and pursues the goal of extracting information from data for decision making, pattern identification, and many other possible applications requiring minimal human interaction. Some examples of ML applications are self-driving cars [1], website recommendation services [2], satellite image recognition [3] and many more [4–9]. For a ML algorithm to learn and achieve a good accuracy, the data used in it needs to have both quality - the training data should be representative of the target application scenario - and quantity - a sufficiently large volume of data should be available to provide an adequate characterization of the phenomenon to be modeled. Since end-users need ML models to be built as fast as possible, the training procedure also demands relatively high resource requirements that scale with the targeted accuracy and amount of training data.

As the amount of digital data grow, novel sophisticated ML algorithms are developed and larger applications for ML are frequently deployed, demanding an exponential amount of resources from users in large scale jobs. Associated to these models are hyperparameters, which is a type of parameter that controls the training process of ML algorithm. To enhance the accuracy of ML model, end users were accustomed to tune its parameters, but the complexity of testing and tuning the hyperparameters of a ML job has become prohibitive given the increasing complexity of the ML jobs being currently used. Therefore, researchers have investigated automated optimization techniques that address hyperparameter selection of machine learning jobs [10–14]. These optimization methods follow a black box approach, which requires testing the model multiple times in different configurations. Given the resource-intensive nature of training and optimizing complex ML jobs [15], users have naturally turned to the cloud to deploy this kind of jobs.

Cloud computing is one of the areas in technology that has bloomed more in recent years, allowing us to offload large workloads to large data centers that have the ability to process them in a relatively short time. Using cloud computing can lead to significant capital cost savings thanks to its convenient pay-

per-what-you-use pricing model. However, given the abundance and heterogeneity of available cloud resources, users are faced with a complex choice when they need to pick the right type and amount of resources for deploying their jobs. Thereupon, researchers developed systems such as [6, 7, 16–19], to perform cloud optimization that enables users to reach a decision for what type of cloud configurations should it pick to perform a certain job.

Unfortunately, though, most of the existing literature looks at the optimization of the cloud configuration for a ML job and at the tuning of the hyper-parameters of a ML job as to two independent problems. Only very recently [7], the importance of jointly optimizing these two types of parameters has been recognized. In fact, the choice of hyper-parameters related to, e.g., the synchronization of the parallel/distributed training process can be strongly affected by the number and type of cloud resources employed to support the training process (e.g., a small cluster of powerful machines vs a large cluster of inexpensive machines). As a consequence, optimizing the two set of parameters (model's hyper-parameters and cloud configuration) independently, as done in most of the existing literature, can lead to identifying configurations that are up to 3.7$\times$ less efficient [7]. On the other hand, joint optimizing these two set of parameters leads to an exponential growth of the resulting search space, urging for novel solutions that can efficiently crawl this search space and minimize the cost and latency of the resulting optimization process.

1.2 Objectives

In the following, the state of the art in the area of optimization of ML training jobs in the cloud is critically analyzed. In the light of this analysis, two main research directions are identified and proposed for my MSc dissertation:

1. Investigating how to extend BOHB [12], a recently proposed method for hyperparameter tuning to optimize, in a joint fashion, both the model's hyper-parameter and the choice of the underlying cloud platform. The key idea at the basis of BOHB is to test configurations "partially", i.e., allocating an intentionally limited "budget" (e.g., time or cost) to each configuration test and timing out the testing once the allocated budget is depleted. This information is used to build a model of the application's efficiency over the set of untested configurations, which can then be consulted to drive the optimization process. The process is then repeated iteratively, invoking the model to select which configurations to test in the next iteration, which will test a number of configurations decreased by a factor β, allocating to each configuration test a budget increased by the same factor β. Unlike in conventional model-driven approaches [10, 13, 20], which do not explicitly control the cost of testing a configuration, the cost incurred to create a model can be significantly reduced.

2. BOHB predicts the quality of a configuration via models that are built considering a specific testing budget. As the optimization process progresses, the budget used for testing increases exponentially, and the number of configurations tested at each iteration also drops with an exponential rate. As a consequence, the models used by BOHB, as the optimization process evolves, are based

on an exponentially decaying number of configurations, which, we argue, can limit their prediction accuracy significantly. To cope with this limitation, we plan to extend BOHB to incorporate transfer learning techniques aimed at extrapolating the predicted configuration quality across different budgets. Through the use of transfer learning techniques, the models used by BOHB to steer the optimization process will be able to retain and exploit the knowledge acquired when testing configurations with smaller budgets.

1.3 Contributions

This thesis focuses on the analysis of machine learning optimization systems, providing insights and developing a system that covers underlying drawbacks of recent state-of-the-art systems that are shown to have good performances. The main contributions are:

- Overview analysis and comparison of state-of-the-art systems.
- Hydra, a self-tuning system solution that performs optimization of machine learning algorithms improving some drawbacks of previous systems by rapidly converging towards the optimum solution without wasting time on bootstrapping the model, using many low-budget evaluations of configurations while applying transfer-learning to enhance the models' performance, ultimately reducing overall costs.

1.4 Thesis Outline

This thesis is structured in five Chapters. Chapter 2 covers the background on this thesis, describing modeling and optimization techniques and will then further analyze related work on systems that optimize machine learning hyperparameters and/or cloud resources. Chapters 3 describe Hydra, going through the implementation, design and workflow of the system. Chapter 4 covers the description of the datasets used in the experiments, and also describe the experiments themselves, and then finally will evaluate the results of Hydra in the previously described experiments. Lastly, in Chapter 5, the conclusions of this work is presented and possible future work in order to improve and innovate the proposed system.

Chapter 2

Background

In this chapter, we will firstly cover some background on black-box modeling and some relevant optimization techniques, followed by the introduction of state-of-the-art systems in the context of hyperparameter optimization. Then, we transition to cloud optimization. Finally, a brief summary of the presented state-of-the-art techniques is discussed.

2.1 Background on black-box modeling and optimization techniques

In the context of optimization and self-tuning of complex applications, the goal is to optimize the application's performance/efficiency while minimizing the number of application's configurations to be tested.

This section overviews existing black-box methods aimed respectively at modeling 2.1.1 and at optimizing 2.1.2 the performance of complex systems.

2.1.1 Black-box modeling for regression and classification

In the following, we provide background on some of black-box modeling techniques that are more commonly employed to capture the dynamics of complex machine learning training jobs. As we will see, in fact, the modeling techniques surveyed in the following are at the basis of the systems, reviewed in Section 2.1.2, which target the optimization of this challenging type of systems.

Gaussian Processes

A Gaussian Process [21] (GP) is a stochastic process that is a generalization of the Gaussian probability distribution. It is a non-parametric approach, which means that it will find a distribution of the possible functions that are similar to the data previously observed.

According to [21] a GP defines a prior over normally distributed functions, and once some data has been acquired, it can be converted into a posterior over functions. Even though it might seem unfeasible

to find a function that correctly represents a distribution, it is facilitated by having the distribution over the function's values at a finite set of points.

Given a finite set of points $x_1, ..., x_n$ a GP assumes that $p(f(x_1), \ldots, f(x_n))$ is jointly Gaussian with some mean $\mu(x)$ and covariance $\sum(x)$ given by $\sum_{ij} = k(x_i, x_j)$, where k is a positive definite covariance function or kernel. The main idea here is that if some x_i is considered to be similar to x_j by the covariance function, then it is expected that the output of the function at those points should be similar as well.

Transfer Learning

In a black-box hyperparameter optimization scenario, we strive to learn what are the specific configurations that maximize the objective function. In order to do that, at the beginning, we need to sample some configurations that will have a relatively high probability of achieving bad results (since they are random) and will have a high cost.

Despite of having tested worse performing configurations, some information can be leveraged in order to improve the probability of selecting a good configuration in the future. If a rough approximation of the objective function is produced, we can take this one step further by sampling from it and take that information into account when selecting a configuration to be sampled in the real objective function. Also, it is possible to extrapolate knowledge from similar experiments so that the initial model is better and improve its overall convergence to the optimum.

This is the base goal of transfer learning, i.e. passing the knowledge from previous tasks that are similar to the real. Since the similarity between tasks can vary, we need to evaluate whether or not it is worth it to use this technique. This is usually done by comparing the base model without training and one with transfer learning training. If the accuracy increases and the training time decrease with transfer learning, it can be advantageous to use.

Some works [22] have done this by learning the model using samples from low-cost approximations of the real model that produce a rough estimate of the performance of the authentic system. In [23], transfer learning techniques are implemented to leverage previous tested models on different dataset to explore hyperparameter settings for a better result.

Recommender Systems

In abstract terms, Recommender Systems (RS) [24] have the goal of providing the end users with recommendations of some object/property that they can find interesting. Essentially, they are useful tools that interact with complex datasets of information, which can deduct the interest of users.

This type of systems are found everywhere on the web, especially e-commerce websites such as Netflix or Amazon and can be advantageous towards both the user and the e-commerce owner. Given that the goal is to provide relevant user-specific recommendations, this can be done if we keep a track of user preferences, e.g. how much did they like a certain movie. Hence, we can send custom suggestions

based on that rating, where the amount of user information that exists on the system is proportional to the accuracy of the produced recommendations.

In the direction of getting the correct recommendations to the correct users, a RS normally uses a rating matrix that keeps users ratings for that purpose. However, this matrix tends to have a large amount of missing values due to the fact that normally users provide feedback on a very limited number of items. To solve this issue, most RS use Collaborative Filtering technique [25] (CF). In order to explain this technique in succinct manner, take the example of two users having a very similar taste for the same set of items; if they have the same utility for items 1 to x, we can safely presume that they should have continue having similar utility with respect to the item $x + 1$. The utility refers to the rating the users provide to items they have bought or had experience with. CF is used by several state-of-the-art systems such as Quasar [16] for instance, that takes advantage of it to estimate the impact of resource scale-out and scale-up, which will be explained in Section 2.3.1.

Decision Trees

Some models that are used for classification and/or regression i.e., their co-domains are respectively discrete and continuous, face common problems such as having many irrelevant predictor variables, therefore needing to preform feature selection improve accuracy. A method that does this quite well is Decision Trees (DT) [26] since they are impervious to the previous problem and offer some advantages such as incorporating naturally numeric and categorical data.

DT builds classification or regression models in a top-down tree-structured fashion. A DT is composed by nodes that have an associated rule, and leaf nodes that label the data. Each node divides the existing data that was transferred from its parent node, according to its own associated rule, and transfer the resulting division towards the two child nodes where on one side reside the data follow the imposed rule and on the other resides the remaining data.

Random Forests (RS) represent an interesting extension of the concept of DTs [27]. RSs are composed by collections of different DT derived from different samples of the dataset and vote on the result of a given input. The outputs of the different DTs, in the inference phase, are then reconciled using a voting scheme. Being build in this manner, RF is less subjective to over fitting than the regular DT [27].

2.1.2 Optimization techniques

Existing optimization techniques can be coarsely classified depending on whether they use a model of the objective-function that is being maximized/minimized or whether they operate in a model-free fashion.

In the following, we overview two model-free optimization approaches, namely hill-climbing and simulated annealing, and one model-based optimization technique, namely Bayesian Optimization.

Model-Free

Hill Climbing and Simulated Annealing

Hill Climbing [28] is a local search technique that can give us a notion about what configuration is better to sample next by moving towards configurations of increasing quality, until it reaches a point where the neighbors of the parameters it chose do not have a higher value. This technique has its key strength in its simplicity. However, it can be severely limited with objective functions that have a large amount of spikes that translate to several local optimums.

Simulated Annealing [29] is another model-free local search technique that improves efficiency significantly regarding previous model-free techniques. It is more robust than hill climbing because hill climbing never picks lower values, i.e. go downhill, hence it might get stuck in a local maximum, and even if it is compared with a stochastic variation of the same algorithm, it proves to be complete but not as efficient.

Unlike hill climbing, simulated annealing can pick bad moves instead of always picking the best. The probability of picking a worse configuration rather than the current best is related to a, so called, ”temperature” variable that decreases over time, where a high temperature increases the probability of a bad configuration being chosen and a low temperature the opposite.

Hence, the general goal in this algorithm is in the beginning start exploring recklessly, e.g. choosing worse configurations than the current ones, and as the algorithm progresses it is more careful to the point where it only chooses better configurations.

Bayesian Optimization

Bayesian Optimization (BO) [30] is a model based optimization technique that operates in an online fashion. More in detail, BO assumes no initial knowledge on the objective function and exploits the information gathered by testing configurations to construct, in an online way, a model of the function being optimized. Overall, BO strives to find a good balance in the exploration vs exploitation dilemma [31] by favoring the selection of configurations that, on one hand, the model predicts to have high quality (thus exploiting the model’s knowledge), as well as, of configurations for which the model shows large uncertainty (thus encouraging explorative behaviors).

As stated in [30] BO uses prior functions, such as GPs as presented previously, to build a probabilistic model of the sampled data. The configuration selection is based on an acquisition functions that choose what point is the best candidate to sample and several acquisition functions have been proposed in the literature. Probability of Improvement (PI) [32] is an acquisition function that considers points that have

a higher probability than the current one of being better. This has the disadvantage of discarding points that might not fit this requirement but have higher uncertainty. Hence, PI is prone to under-explore globally and getting stuck into local optima [30]. Expected Improvement (EI) is another acquisition function that, analogously to PI, looks for configurations that are likely to outperform the currently known optimum. However, unlike PI, the EI uses the model to estimate the predicted magnitude of improvement with respect to the current optimum. As such, EI tends to achieve a better tradeoff between exploration and exploitation.

BO algorithms have many variants and the main differences between them is in the methods they employ to construct a model. In section 2.2.1 we will discuss BO with GP as a model in an hyperparameter optimization context.

2.2 Hyperparameter Optimization

As previously discussed, in machine learning many algorithms require the user to set some *hyperparameters*. This type of parameters are called hyper because they influence how the algorithm will learn. Examples of hyperparameters are the synchronization method and batch size used by different workers in a distributed training process [33]. Unfortunately, guessing a good value for a model's hyperparameters beforehand is far from being a trivial task, as their correct tuning is affected by a large number of factors, such as the shape of the function that the ML model is learning or the number/type of computational resources being harnessed in the learning process.

In the section 2.1.2 we have covered some approaches for optimization can be applied in this scenario. However, machine learning algorithms tend to have large training times and require a large number of resources such as powerful CPUs and in some cases even one or more GPUs. Given this, it is imperative that the optimization task minimizes both cost and time while providing a set of hyperparameters that ensure optimal (or close to optimum) performance.

In this section we will review some state-of-the-art optimization techniques in the scenario of hyperparameter optimization that approach the problem in different ways.

2.2.1 Bayesian Optimization

The existing approaches that use BO for hyperparameter optimization build a model, often based on GP, that predicts, for each possible hyperparameter value, the corresponding accuracy achievable by the model. A great advantage of this model is that it accumulates all data from previous evaluations of the objective function, which typically leads to producing more accurate predictions and, consequently, to enhance the speed of convergence towards optimal solutions.

This technique proves to be very efficient in providing highly accurate configurations but there are some intrinsic weaknesses associated to it, such as: *i)* it needs to have some samples before building a model; *ii)* each individual sample has a high cost; *iii)* GPs are very slow to train, especially if a large number of configurations have been tested. Consequently the initial phase is costly and slow. Besides

this, in large datasets or in scenarios with a substantial amount of hyperparameters, BO will scale poorly because it will need to train the algorithm on the whole dataset and build increasingly complex models for each hyperparameter it is added.

In the following, we will discuss some algorithms that mitigate this disadvantage by adopting techniques such as transfer learning and others that provide cheaper costs and meet the same or better results in the process.

2.2.2 Hyperband

Hyperband [11] (HB), is characterized as a model-free technique of hyperparameter optimization that originated from pure exploitation bandit problems, that have the goal of minimizing *regret*, that is the distance from the optimal solution as fast as possible, in any setting. This algorithm extends the Successive Halving [34] (SH) algorithm that performs in the following way: Given a growth factor *N*, a minimum and maximum *budget* (e.g. wall clock time), a fixed number of randomly sampled configurations will be evaluated with the minimum budget, then they will be compared. The top *1/N*, multiplied by the number of tested configurations will pass to the next phase, which is similar to the previous phase, with the difference that it only evaluates the passed configurations and having the budget multiplied by N. This algorithm stops when the maximum budget value is reached.

As described in [11], low budgets will produce noisy evaluations that can be misleading in SH, so HB tackles this concern by doing multiple runs of SH and increasing, at each run, the minimum budget. As such, HB mitigates the risk of being biased, hence staying in the Successive Halving iteration. However, it may not scale well when the budget increases. Ultimately, HB recommends the configuration that performed best across every run of SH.

HB has the advantage of being very fast regarding proposing good configurations in early stages. By comparing it with BO, it proposes configurations and converges faster in the early stages. However, due to its stochastic nature, HB suffers from the same issue of Random Search [35]. For instance, HB does not leverage the information of previously done evaluations, since it only maintains a record of the best performing configuration, making it converge slowly and most likely not reach the global optimum.

2.2.3 Fabolas

Fabolas [36] is a state-of-the-art technique for hyperparameter optimization that tries to increase the efficiency of BO when used to optimize machine learning jobs that need to digest large datasets. The idea at the basis of Fabolas is to infer optimal configurations for training using the full dataset, based only on observations performed using a subsampled dataset. This leads to speeding up the initial model building phase and provides faster results when compared to traditional BO-based hyperparameter optimization techniques [10, 13, 20].

Producing a model while using a subsampled dataset will result in cheaper function evaluations, however it will also produce a worse approximation of the objective function which in turn will provide worse samples. To tackle this issue, Fabolas models accuracy and training time as a function not only

of the hyperparameters' configuration, but also of the dataset size. Based on this model, Fabolas seeks the best trade-off global optimum. Finally, Fabolas extrapolates the knowledge to the original dataset by predicting what configurations will achieve the best result.

This approach of constraining the resources needed to build the model in order to lower the cost of function evaluations is similar to the HB approach of doing *budget* runs to minimize cost. However Fabolas is not as fast as HB in the initial phase but it does converge faster than BO in general.

2.2.4 Google Vizier

Google Vizier [13] is a service that provides distributed black-box optimization techniques and can, as such, be applied not only to hyperparameter optimization but also to other domains such as automated A/B testing e.g. tuning user–interface parameters. Vizier has then been superseded by Google's AutoML, which Google offers as a commercial service aimed to simplify the development of machine learning-based services.

In Google Vizier, a study is an optimization process, that is composed by a set of trials. Each trial is essentially the function evaluation of the algorithm to be optimized with a set of hyperparameters. Its workflow is composed by suggestion workers that run the trials of a study, then they post the result in a persistent database. After posting the result, evaluation workers will analyze the trial results and then request new trials for the system with new parameters, which will get forwarded to the suggestion service, that is composed by suggestion workers and will repeat the process. Each suggestion worker will get a distributed lock while working on a trial, and will release it and write to the persistent data base once it has completed or if it had a problem, e.g. has crashed. This enables the system to detect faulty studies and preempt the study to stop wasting resources. Vizier also provides a mechanism that enables the client to implement its own optimization algorithms instead of the ones provided by the platform.

The system has an early stopping mechanism to stop trials that are expected to have bad outcomes. In order to do this efficiently, Vizier has two rules that need to be valid in order for the Trial to proceed; the first one is the performance curve stopping rule, that will halt the trial if the performance at a given time is falling short. This is accomplished by keeping track of the performance of previous trials of a study as well as a set of measurements taken during the trial evaluation, then model this data to a Bayesian regressor and compare the results. The second rule is the median stopping rule, that will stop a trial at a given step if its objective value is inferior than the median objective value of all previous completed trials in that study up to that step.

Besides having mechanisms that improve scalability of the system, Vizier also improves its optimization efficiency by applying transfer learning to improve configuration selection of the trails. It leverages obtained information form previous studies by using GP regressors stack, where each regressor is associated to a study, and all of the regressors are trained on the residuals relative to the regressor below it, so the top-level regressor transmits data of all studies that were made that can aid the optimization process. However in order to perform transfer learning it has heavy assumptions about the underlying

performance for the same dataset in different algorithms, and furthermore transfer learning in this system is only valuable for studies that have many trials, which is something that is preferred to be as small as possible for a model-based ML algorithm.

2.2.5 BOHB

BOHB [12] is a state-of-the-art technique that combines two techniques that were previously discussed, namely Hyperband [11] and BO [30]. It does so in order to leverage the advantages that both algorithms bring, while minimizing their disadvantages.

BOHB performs BO with a different modeling scheme, instead of using GP to model the objective function, it uses a Tree Parzen Estimator [30] (TPE). TPE uses a kernel density estimator which instead of modeling the objective function directly as GP does, models two different densities over the input configuration space. These densities are represented by $l(x)$ and $g(x)$; where the first density captures configurations that performed significantly well, that are above a certain α threshold, and the second has configurations that have undesirable results, that are below the α threshold. They can be represented by:

$$l(x) = p(x < \alpha | x, D)$$

$$g(x) = p(x > \alpha | x, D)$$

This change of model came due to the fact that TPE scales better than GP, while maintaining the support for mixed discrete and continuous configuration spaces. In order to sample with TPE, the new configuration space to be evaluated is the one that maximizes the ratio $\frac{l(x)}{g(x)}$ which is equivalent to maximizing the EI.

The BOHB algorithm starts after the user has defined a minimum and a maximum budget , where budget can represent any type of constraint on computational resources, such as execution time, epochs, and others. After declaring the budgets boundaries, the algorithm will start by doing the Hyperband method with a relevant difference: unlike HB, BOHB does not sample configurations randomly. Instead, it does not always randomly sample configurations; conversely, it relies on the TPE-based models, constructed in previous HB runs, to determine which configurations to test in the next HB run. This way, the knowledge acquired by testing configurations in previous HB runs is retained and exploited to drive the future HB runs and enhance convergence speed.

This method has the speed advantage of Hyperband, i.e., it is able to reduce the cost of evaluating the quality of configurations by controlling the computational budget allocated over time to function evaluation. However, thanks to the use of a model, it preserves BO's effectiveness in guiding the search process towards global optima. Additionally, BOHB selects, with a small, user-tunable probability, con-

figurations in a purely random way (i.e., without consulting the model). This design choice improves the robustness of BOHB in presence of inaccurate/flawed models, which, in pure model-driven approaches (e.g., based on BO [10]) are likely to hinder the efficiency of the optimization process.

2.2.6 Efficient Transfer Learning Method for Automatic Hyperparameter Tuning

This work [14] focuses on leveraging information acquired via optimization of machine learning algorithms in different datasets to improve the overall accuracy while minimizing the overhead of performing such techniques. It is an instance of BO[10] that can perform transfer learning by building a common response surface on all evaluated datasets. BO is a framework that is used in this scenario because it takes the advantage of performing cheaper evaluation on approximations of the objective function by constructing a surrogate that uses, in this case, EI as the evaluation criteria and GP as its model.

The present method builds a common response surface as [37] with the difference that instead of using a ranking surrogate to approximate the surface, it uses the mean value derivation per dataset that reduces execution complexity, which is crucial, considering that these algorithms are executed in each evaluation stage.

Unlike traditional SMBO methods, instead of modeling in each new dataset the surrogate GP to a new unknown function directly, it uses the derivations of the per-dataset mean value as the common response surface values, and in each iteration of function evaluations, the response surface changes based on the output value of that evaluation and the mean and standard deviation values of all the function evaluations up to that point of that dataset. The following function as denoted in [14] explains this, where y is the response value that is used by the GP, t represents the iteration number of function evaluations, d is the dataset being used and finally μ and σ are the mean and standard deviation values respectively.

$$y_t^d = \frac{f^d(x_t) - \mu^d}{\sigma^d}$$

This method is valid if we assume that different datasets produce resembling functions without taking into account the scale and location difference between two functions. So, using the derivation of the dataset mean as the response surface is valid as it uses GP models with different means for different datasets. However this assumption can be sometimes unrealistic and can possibly lead to a worse performance in datasets that produce dissimilar functions. In addition, the use of transfer-learning in this method requires some *a priori* knowledge that can be obtained only after performing some function evaluations. As such, this method scales slowly at first and is only be able provide significant improvements after gathering a substantial amount of knowledge.

2.3 Optimization in the Cloud

This section reviews a set of state-of-the-art approaches that tackled the problem of optimizing, according to different metrics, the efficiency of complex applications to be deployed on the cloud. As it will discussed, most of the solution in this area of the literature treat the application as a black-box and focus solely on the identification of the right amount and type of cloud resources to be allocated to the application to meet user-defined constraints on QoS. Examples of this type of approaches can be found in Sections 2.3.1-2.3.5.

A notable exception is Lynceus, which instead optimizes the application and cloud parameters in a joint fashion (see Section 2.3.5). However, this introduces a plethora of decisions and problems users face, such as, what type of instances should be rented in order to perform this task, how many virtual machines should be rented, which one is more suited for a specific workload and how much money do they cost. Pairing these issues with machine learning optimization, even experienced users might be overwhelmed by both the difficulty and amount of decisions they need to make for the service to be worthwhile.

In this section we will introduce some state-of-the-art systems that provide solutions to this type of problem.

2.3.1 Quasar

Quasar [16] aims at determining the cloud resources needed for each workload, which is done by profiling a workload when it arrives. The workload is classified by offline knowledge, that is gathered by observing previous workloads. Quasar characterizes a workload according to four main dimensions: resources per node and number of nodes for allocation, number of servers, heterogeneity (types of servers used) and interference.

After profiling the workload, the system compares the profiling results of the analyzed workloads with current available labels to conclude the characterization process. This is done by using CF [25] techniques as mentioned in section 2.1.1.

Quasar is similar to another state-of-the-art system, named Paragon [38], since it has the same goals and applies CF techniques to profile received workloads, however Paragon solely evaluates resource assignment, so it only is able to characterize workloads with respect to interference and heterogeneity i.e. server type. By handling both resource assignment and allocation, Quasar also characterizes scale-out and scale-in for each workload which causes the space of assignments significantly larger. This can be advantageous as it explores more options for allocation but it can quickly turn the space of possible allocations to be gigantic.

To cope with this problem, the CF techniques that are used to assign the workload to the available machines follow a policy of minimum resource allocation per job, in which the resources that are examined first are the largest ones. That way, Quasar minimizes the amount of combinations of resources that satisfy the constraints, effectively reducing the search space.

As the system receives more workloads and uses CF techniques, it learns more about them and improves classification performance over time which directly improves overall performance, as well as maximizing resource utilization, reducing user expenses while achieving quality of service constraints.

Despite being a cost efficient system, Quasar suffers from only optimize cloud parameters, not application ones. It also need knowledge of previous workloads/offline data to function efficiently and disregards optimization costs.

2.3.2 HCloud

As previously mentioned, cloud computing provides both flexibility and high performance for users while also being cost efficient to operators, but it can prove challenging for users to choose what instance size, what type, if they are short-term or long-term allocations. Some strategies try to solve this multi-dimensional problem but overlook differentiating reserved and on-demand provisioning strategies.

HCloud [17] is a state-of-the-art hybrid provisioning system that leverages both reserved and on-demand resources, which are resources that are reserved and payed for large in large periods of time e.g. one year, and resources that are provided whenever the user desires and payed for by each second/minute of utilization, respectively. It does so to minimize cost and maximize resource utility in the cloud. For every job, it determines what type of provisioning strategy is more advantageous based on the load and resource unpredictability of an instance. It factors in the decision making process the cost, performance unpredictability, instantiation overheads and provisioning flexibility of an instance, while it is aware of the quality of service it needs to maintain and maximizing resource efficiency.

By taking advantage of on-demand and reserved resources, higher resource efficiency can be achieved for the end-user, however traditionally the user always needs to specify how many resources each job uses, and this decision is prone to be erroneous and leads to over-provisioning. In order to minimize this issue, HCloud uses Quasar's [16] estimators of resource preference, as well as interference provisioning estimators, to improve provisioning accuracy. The interference provisioning estimators play a great part in this system, as on-demand resources tend to have great external interference, having more the smaller the instance is.

To decide how to map jobs to resources of different types, a policy was devised that was based on three principles; Reserved resources are always consumed first rather than on-demand; Jobs that are mapped to on-demand instances cannot delay interference sensitive jobs; Utilization of reserved instances are to be carefully managed in order to reduce queue.

Even though this system improves on others regarding the use of provisioning strategies, it is still focused on providing resource efficiency, ignoring allocation and deployment costs.

2.3.3 CherryPick

Picking the best performing cloud configurations for a job has been a challenge and many previous strategies such as [16] leveraged recommender systems techniques line CF and/or applied decision policies to maximize resource utilization [16][17].

CherryPick [18] is a system that instead leverages Bayesian Optimization [30] (BO) to build a predictive model that is used to find cloud configurations that not only guarantee application performance, but also minimize cloud usage cost.

This approach views the problem from a different perspective but it still faces daunting issues regarding cloud configurations. The first issue is focused on the cloud computing performance model. It is quite complex, because the amount of time that takes a resource to complete a job is affected by its configuration in a non-linear manner, e.g. two configurations might take the same time executing a job, even if the second configuration had double the RAM amount. The second problem is that, while cloud providers charge users based on the time the virtual machines are executing and more powerful machines are more costly and execute faster, it is hard to find the balance between a fast expensive machines and a cheaper slower machines. Heterogeneity of applications is the third and final major issue, since multiple applications can have significantly different performances with slightly different cloud configurations.

As to tackle this obstacle, CherryPick uses a objective function that is optimized by the system. It minimizes the deployment cost of a given configuration in function of the time the machines are up and how much they cost per hour, added with a time constraint to guarantee application performance. Similarly as previous techniques that leveraged BO, CherryPick uses GP [21] to build the model as well as EI_c (constrained EI), as it needs to factor in the probability of meeting the execution time constraints.

CherryPick begins by sampling three quasi-random points in order to retrieve an estimate of the cost function that is to be minimized. Then, regular BO is performed, choosing the best points to be sampled that maximize EI.

CherryPick has two main drawbacks. First, it only aims at optimizing the cloud configurations, neglecting the fact that in many applications (in particular, distributed training of ML models) optimal efficiency can only be attained by jointly optimizing both the cloud configuration and the application's configuration. Second, being based on BO it suffers from the same shortcomings already discussed in Section2.2.1 in the context of hyperparameter optimization. Namely, it requires an initial bootstrapping phase that, due to its random nature, may lead to testing very expensive configurations. Furthermore, in order to assess the quality of configurations, it can require applications to execute in each configuration for a long period of time (thus incurring a large economical cost).

2.3.4 PARIS

As CherryPick, PARIS [19] is another state-of-the-art technique that has a different approach on solving the cloud configuration selection problem. It provides performance estimates with minimal data collection and uses random forests, as described in section 2.1.1, instead of BO [30].

After the user provides a representative task of the workload, as well as indicating the desired performance metric and set of candidate virtual machine types, PARIS proceeds by outputting cost and performance predictions of the instances provided by the user. In order to make such predictions, it needs to know the resource requirements for the workload and how the indicated instances affect it.

Instead of executing the workload on all of the machine types, the modeling approach of PARIS operates in two stages. The first of them is an offline stage that will run one single time and has the objective of benchmarking various workloads with each machine type, providing detailed metrics such as CPU utilization,network utilization, disk utilization and memory utilization. This stage will only need to run whenever a new instance type that was not bench marked is added, as to minimize repetitive work. After benchmarking the virtual machines, a set of decision trees are trained for each workload and will eventually build a forest. The second is the online stage that will run the task provided by the user on two, pre-defined, reference virtual machines in order to collect information on the corresponding usage of resources and performances.

After the forest have been trained according to user specified performance metrics, the information that was acquired in the online stage is fed to the forest. The forest will finally output the mean and the $90th$ percentile performance for that machine type and will repeat after doing the same process for every candidate virtual machine types.

Now, since resource requirements for the workload are known and how the indicated instances affect it, PARIS can build a performance-cost trade-off map after estimating the cost of each instance, which is made by assuming cost is a function of the provided performance metric and the cost per hour of instances, that is assumed to be known as well.

This map aids the user in choosing instance types significantly, however PARIS accuracy is strongly dependent in the choice of reference configurations as well as the quality of the data in the training set. Furthermore, PARIS requires *a priori* knowledge of some similar workload to work. In fact, if the set of workloads used in the first stage are not representative of the application to be actually optimized, PARIS' accuracy might be severely compromised.

2.3.5 Lynceus

Lynceus [7] is another recent approach for the optimization of cloud-based jobs. It adopts model-driven optimization as [18] and [19] but it refines its model in a different manner. Lynceus is a budget-aware and long-sighted self-tuning system of cloud resources that has the goal of discovering the configurations that minimize the execution cost of data analytic jobs by ensuring that the maximum execution time constraint is followed and the evaluation of a configuration does not exceed a given budget, where a configuration in this scenario is composed by cloud parameters (e.g. virtual machine type and number of instances) as well as hyperparameters of machine learning jobs.

To achieve its goals, this system has the following strategy: At the beginning of the exploration phase, where it strives to find a good configuration and there is a large uncertainty in the cost model, Lynceus allows for a larger budget and presents a more explorative behavior. As the system explores more configurations and the cost model becomes more accurate, the budget will decrease. In this phase, Lynceus adopts a more careful and exploitative approach where it only selects configurations that will

not compromise the given budget, while leveraging the cost model to achieve the maximum shorter reward.

It uses BO [30] to solve the optimization problem, with the acquisition function of Constrained Expected Improvement [39] (EIC). A bagging ensemble of decision trees [40] are trained with a dataset containing the tested configurations and used to build the cost model. This latter technique is similar to random forest as described in Section 2.1.1 but does not integrate the random selection of subsets of training data and uses all available features for each tree.

Lynceus also leverages a look-ahead technique that allows to foresee the effect of choosing a configuration to sample. It predicts a path of configurations to be explored by simulating evaluations using predicted values by the model. The model will then be updated with the predicted cost of the configuration which has the highest reward and will predict a reward and cost value for configurations that have not been tested yet. For each evaluation iteration, it computes the reward of configurations that are assumed to have a 99% probability of having a lower cost than the current budget and then based on the current model, it predicts the improvement over the current know best configuration. The configuration that maximizes the reward is used to update the model.

EIC is used due to the fact that Lynceus aims to ensure that the selected configuration will not only minimize cost but will also meet user-defined constraints on the jobs. EIC of a certain configuration is the product of the standard EI [30]of a given configuration and the probability of that point it respects the constraints.

Lynceus does consider both cloud and application's configuration parameters jointly. However, due to its reliance on BO, it suffers of the same problems already discussed when introducing CherryPick, which are reacquiring an initial bootstrapping phase that, may lead to testing very expensive configurations.

2.4 Summary

This section has provided an overview of some of the most relevant state of the art techniques in the context of hyperparameter optimization and optimization in the cloud. Important background on general optimization techniques was also discussed, giving us insight about possible paths that may help achieve the objectives stated in 1.2.

Table 2.1 outlines important characteristics concerning state-of-the-art techniques covered in this report. It compares base optimization techniques, optimization models, what parameters the system takes into account, if it applies transfer-learning to extrapolate information obtained from partially observing configurations or from previously observed workloads in the optimization process, and finally if it requires offline knowledge to do so.

System	Base Optimization Technique	Modeling Technique	Optimization Parameters	Transfer-learning	Requires Offline Knowledge
BO	Bayesian Optimization	Gaussian Process Regression	ML Algorithm Hyperparameters	No	No
Hyperband	Pseudo-random Sampling	None	ML Algorithm Hyperparameters	No	No
Fabolas	Bayesian Optimization	Gaussian Process Regression	ML Algorithm Hyperparameters	From partially sampled configurations	No
Google Vizier	Bayesian Optimization	Gaussian Process Regression	ML Algorithm Hyperparameters	From other workloads	Not needed, but can be
BOHB	Bayesian Optimization	Tree Parzen Estimator	ML Algorithm Hyperparameters	No	No
Efficient Transfer Learning Method	Sequential Model-based Optimization	Gaussian Process Regression	ML Algorithm Hyperparameters	From other workloads	No
Quasar	Policy-based Optimization with Collaborative Filtering	Collaborative Filtering	Cloud parameters	From other workloads	Yes
HCloud	Policy-based Optimization with Interference estimators	Collaborative Filtering	Cloud parameters	From other workloads	Yes
CherryPick	Bayesian Optimization	Gaussian Process Regression	Cloud parameters	No	No
PARIS	Offline Online stage Optimization	Random Forest Regression	Cloud parameters	From other workloads	Yes
Lynceus	Bayesian Optimization with Lookahead	Bagging Ensemble of Decision Trees	ML algorithm Hyperparameters and Cloud parameters	No	No

Table 2.1: Comparative analysis of state-of-the-art systems.

Overall by the analysis of the table we get that:

1. Only Lynceus aims at optimizing both cloud and application parameters. The authors of that solution have also reported experimental data that confirms the relevance of optimizing these parameters in a joint fashion, with gains (in terms of cost reduction for the users) that can extend up to a factor 3.7$\times$ when compared to solutions that optimize the two set of parameters independently.

2. Unfortunately, the reliance of Lynceus on BO [10] exposes it to a number of shortcomings that have been highlighted by the recent literature on hyperparameter optimization. More in detail, In BO each function evaluation is very expensive and it requires model bootstrapping time in the beginning. Overall, these aspects can make BO prohibitively expensive if the jobs to be deployed demand a large amount of computational resources. BOHB overcomes these issues, and appears, as such, to be a very promising solution. However, BOHB has never been applied to jointly optimize cloud and hyperparameters of a ML job. Another limitation of BOHB is that it relies on models built considering only the information acquired when testing configurations with the largest budget tested so far. As the number of tested configurations drops exponentially with the available budget, the models used by BOHB are prone to suffer from data scarcity.

3. There are techniques that do not require to build a model, hence, in that short time window, they gain some benefit for those resources, however they are quickly outclassed in terms of convergence to the global optimum after some time. Hyperband [11] is one of those techniques, but despite its convergence rate being sub-optimal, for short-time optimizations it can produce great

results, taking advantage of its many low-budget executions that produce configurations that are accurate enough to be considered.

Chapter 3

The Hydra Optimizer

In this chapter we propose and cover the design/implementation of Hydra, a system that build on BOHB and extends it to address its main shortcomings, such as the inability to extrapolate how the quality of configurations vary across budgets. We also present variants of this system that try to balance the economic cost of the optimization process by taking it into account throughout the run.

3.1 Overview

Performing optimization of machine learning algorithm is an expensive procedure. As a matter of fact, even with BO-based techniques [10], which strive to minimize the number of evaluations needed to reach a global optimum, each evaluation can still be very demanding in terms of resources and time. Other techniques, such as Fabolas [36], have addressed this problem by using sub-sampling in the training dataset so as to reduce cost of evaluating the quality of configurations during the optimization process.

These techniques require a model to be built firstly in order to produce results that can lead them to the global optima and since this task is done by randomly sampling some configurations there is always a fixed amount of resources that is spent and do not contribute to the end goal in that time window. This said, BOHB [12] leverages both Hyperband [11] and Bayesian Optimization to produce results before the model has been built and after, provide more information to the model with the configurations that have been evaluated. This allows for achieving convergence rates that are faster than BO, while constraining evaluations to a budget, so as to reduce the cost of the optimization process. BOHB then effectively counters the presented issue with traditional BO. Furthermore it inherits some beneficial features of Hyperband in the sampling phase, since it has the possibility of exploring random configurations, while reducing the penalty of it being sub-optimal via the use of the Successive Halving [34] technique.

Extending BOHB to jointly optimize both the ML applications and the cloud configuration is not trivial. The first challenge with BOHB is understanding what is the most efficient way to employ BOHB for this purpose. One key question that arises is whether the cloud configurations should be treated in an opaque way i.e. similar to additional hyperparameters in a hyperparameter configuration. The risk

of such a simplistic approach is that it exposes to the risk of sampling very expensive configurations that require a large amount of computational resources unnecessarily, e.g., in the initial phases of the optimization process where no or very little knowledge is available on the job being optimized.

Another shortcoming of BOHB that we intend to address is its inability to exploit information gathered when testing configurations with small budgets. In order to overcome this limitation, we plan to use transfer-learning. By recording the performance of various configurations evaluated using diverse budget levels, Hydra can leverage that information to find a trend between budgets and use it to predict the performance of configurations on larger budgets.

3.2 Design Details

Hydra is a solution that extends BOHB [12], which itself its an extension of Hyperband [11]. Since BOHB proved to possess the speed of Hyperband while being more likely to select high quality configurations via model-based techniques, we argue that by having a similar system with a richer model and a more effective way to extract the model's knowledge (via the use of alternative acquisition functions), one can further enhance the efficiency of the optimization process.

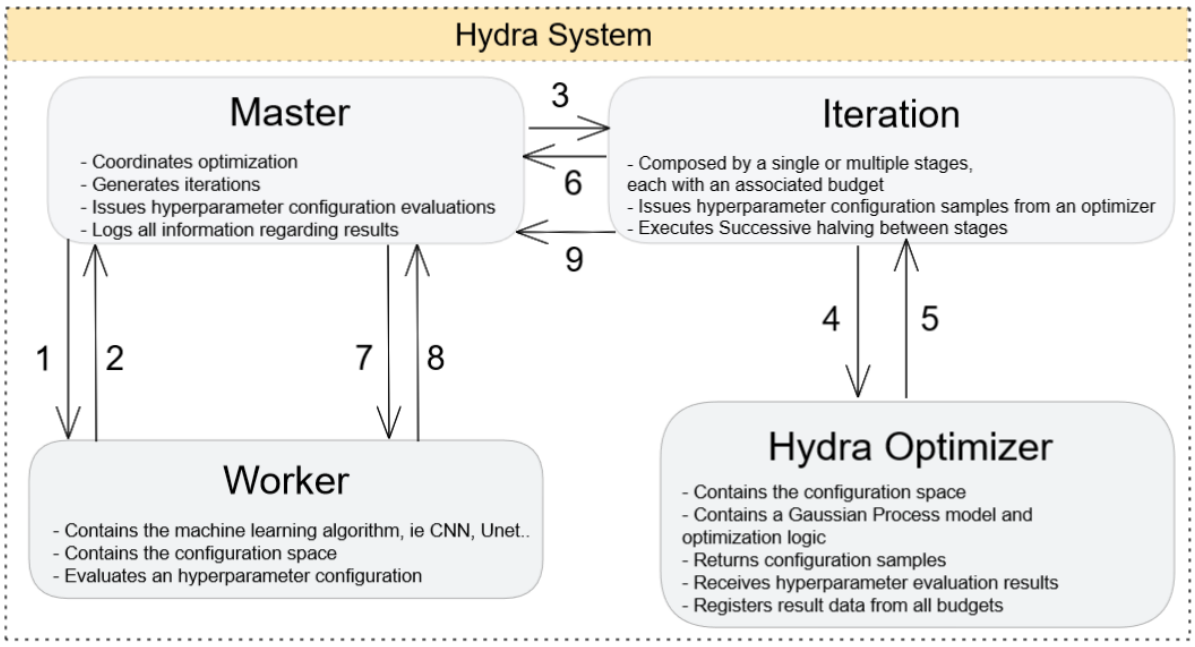


Figure 3.1: Hydra system overview

As showcased in figure 3.1 the system has the following steps: Firstly the Master module will initialize the optimization, generating the intermediate budget values from the given minimum and maximum values. Then (1) it will create a worker with a given objective function i.e. machine learning algorithm and the associated configuration space, containing all different combinations of hyperparameters. Once that is completed it will return the configuration space of the hyperparameters (2) to the master module, where it is shared with the optimizer component. After that, while the number of maximum finished

iterations completed, (3) it will create a new iteration module with the associated initial budget. This module will issue hyperparameter configuration samples from the optimizer (4) with the corresponding budget. Once the optimizer returns a hyperparameter configuration (5), the iteration module will prompt the Master module to queue a run with the given hyperparameters and budget (6). Once the Master receives this request, (7) it will give the Worker a configuration to evaluate with a budget, and, when complete, (8) it will return the accuracy value that was achieved. This value is then registered and passed to the Iteration and Optimizer. The steps (4) to (8) will then be repeated until the Iteration has no more configurations needed to be sampled. After that, it will perform successive halving and advancing a stage, performing steps (6) to (8) with the configurations that have passed but with a higher budget value. Finally, once the iteration finishes its last stage, (9) it informs the Master module that the current iteration has finished. This module then will increase the number of finished iterations and initial budget value, repeating step (3).

In Hydra, we have selected Gaussian Process [21] as base ML technique, since they are the most frequently used models in Bayesian Optimization due to their ability to provide smooth and accurate uncertainty estimates. Specifically, we use Gaussian Processes with Matérn $5/2$ Kernels [41], which is also a common choice in the Bayesian optimization literature given that it produces less restrictive smoothness assumptions [36] — an important feature, given that we plan to add another dimension to the model's feature space, namely the Hyperband budget. Hydra supports various EI-based acquisition functions, including novel ones defined for being used in the context of the HB optimization method.

While performing the optimization, Hydra will gather relevant data from the experiment, namely the configuration that was sampled, the budget that was used, wall-clock time, optimization result, and other information. When using the model to do a prediction, it will use the information it has gathered to train the model, more specifically the configurations it has sampled, their results and their budgets.

BOHB only trains configurations using a single budget, i.e. the largest budget for which a minimum pre-determined number of configurations has been gathered. As a result, only a subset of the available info is exploited whenever the model is used/queried. This problem is solved in Hydra by considering budget as an extra feature and training a single model with data gathered using diverse budgets, in order to enable the construction of models that can extrapolate the trends that arise when the budget varies. Recall also that the objective is to find a configurations that has maximize accuracy using the **full** budget, so the models should be used to identify configurations that will excel at full budget, but that will be at least initially evaluated with lower budget, based on the SH algorithm.

In Hydra, there are some parts of the system that we chose not to change, having the similar behavior as BOHB, such as instead of always using the model to predict a configuration, we still use a probability of sampling instead a random configuration, thereby maintaining Hyperband theoretical guarantees. When producing a prediction, Hydra uses the Hyperband algorithm to determine how many configurations will be generated. If at least $(d + 1)$ configurations (where d is set to the number of dimension in the configuration space) have been evaluated, the model is used to make a prediction on a configuration. Otherwise, we sample configurations randomly according to a uniform distribution. Even

if there are enough results, there is always a probability (which as in BOHB we set to 33.3%) that we chose to sample randomly.

The process of producing a prediction with a model implies: (i) to train the model with all the collected data from the evaluated configurations, and (ii) identify the *incumbent* configuration, i.e., the one that the model predicts to yield the best result (e.g. highest accuracy, lowest loss) when deployed using the full budget. The main reason behind focusing on choosing incumbents that have the highest budget is to guide the model to focus on achieving the best performing configurations on the maximum budget, where there is a higher probability in sampling better configurations. However, by doing so, in scenarios where there are no sampled configurations that have as high budget value, such as the beginning of the first Hyperband bracket, we drop that constraint and allow the incumbents budget to take the value of the highest sampled budget from the gathered data. To collect more information about Hydra behavior, we have also implemented multiple variants that have a different take on how the predictions are made, and in the following sections we will explain what are the main differences and their purpose.

3.2.1 Budget Sampling

Analogously to Hyperband and BOHB, in Hydra the notion of budget can mapped to different metrics that constraint the amount of resources consumed when training a ML model, e.g. wall-clock time, iteration, algorithm epochs, cloud cost.

In both these systems, the algorithm starts by sampling a number of configurations randomly, and evaluates them using the lowest possible budget. In Hydra we use the same procedure, but, as in BOHB, at some point, when enough configurations are gathered to build a model, we can start choosing which configurations to evaluate by leveraging its knowledge. To get a prediction from the model, we go through the search space and for each different configuration (or for a set of randomly chosen configurations if the configuration space is too large to be exhaustively sampled) we compute the Expected Improvement [39] (EI). Finally, we select the configuration that has the highest EI value.

Differently from BOHB, though, in Hydra we treat the budget as a model's feature. When querying the model, we need therefore to establish what value of the budget to specify.The Hyperband algorithm only determines which configurations to evaluated at the beginning of a bracket. From that moment on, the top configurations will pass to the next stage, to be evaluated with a higher budget. This Successive Halving [34] will happen until the maximum budget is reached and the highest quality configuration *using full budget* is returned. Thus, one may argue that the model should be queried to identify the configuration that will achieve maximum quality (e.g., accuracy/loss) in the maximum or final budget. However it is also arguable that what matters in a stage of a bracket is the performance of the configuration on the current budget of that stage. Indeed, a poor performance in early stages would reduce the odds of that configuration to be among the top ones that proceed to the following stages with higher budgets. Consequently we consider two variants: one that selects configurations according to their EI value considering full/maximum budget, which we call Full-Budget Sampling (FBS), and another variant that

similarly selects configurations according to their EI value, but considering the current stage budget. We name this variant Current Budget Sampling (CBS).

The main advantage of the FBS strategy is that it will favor configurations that excel on higher budgets. However, if a configuration does fall short in its performance in lower budget values, it may not be tested in higher budget levels, since Successive Halving may discard that configuration. Conversely, CBS will focus on picking configurations that excel on the lowest budget of the current bracket, thus improving the odds that a model chosen configuration passes through the initial Successive Halving pruning of configurations. Clearly, these two variants will have the same behavior when predicting configurations on a bracket that has as initial budget value the maximum budget value of the experiment.

3.2.2 Cost of evaluating configurations

In the cloud different choices of type and numbers of virtual machines yield different costs. Hydra keeps the cost factor into account by incorporating several cost-aware acquisition functions, which will be evaluated in the following chapter. Expected Improvement per dollar is a classic technique to keep into account costs in BO. However, Hydra introduces a new cost-aware acquisition function tailored for operating in a successive halving scheme. By Expected Improvement per dollar with the Budget Sampling variants discussed in section 3.2.1, with FBS we can sample configurations that have the potential of achieving a high accuracy in higher budget values while possessing a low economic cost. This may hamstring FBS capacity of providing good configurations in low budget value scenarios even more, however we can ensure that whenever a configuration that was sampled through the model, if it reaches the highest budget value stage, it will have a reduced cost. On the other hand, if we consider CBS with this variant, it will sample very economic and well performing configurations in the initial budget value of a bracket, but on the higher budget value stages the configuration might achieve higher than expected costs. Much like in section 3.2.1, both FBS and CBS with this variant will also have similar behavior when sampling configurations in brackets where the initial budget value is the same as the maximum budget value allowed in the experiment. This economic-cost reducing variant may have a cost-reducing prospect, however we need to take into consideration that duplicating a model that is already considered slow compared to Tree-structered Parzen estimators as shown in BOHB [12] may deter the algorithm from being fast and by consequence proving to have a higher economic cost. When we are combining the economic cost variant Expected Improvement per Dollar with budget sample variants FBS and CBS, we are restricting both base model and cost model to have the same *target* budget, e.i. sample according to maximum or current bracket budget. In order to have a prediction that can leverage the performance of configurations on higher budget values while reducing the cost of configurations on current budget values, we developed another variant called Hybrid Sampling. Essentially it uses FBS to retrieve the Expected Improvement of a configuration the highest budget possible, while dividing the cost of the same configuration but on the initial budget a bracket. This variant is expected to propose configurations that have low costs in low budget values and have great performance in the maximum budget value. Since Hyperband proposes a large quantity of configurations in lower budget values, we

expect this to minimize the economic costs greatly in early stages while providing the ability to outperform others in the last.

Data: random run probability p, budget b, maximum budget B, current number of observations O, number of hyperparameters D

Result: next configuration to evaluate

if $O <= (D + 1)$ *or* $random() < p$ **then**

    return random configuration;

end

$best\_configuration$ = None;

$best\_value$ = 0;

for *configuration* c *in the search space* **do**

    ei = compute the EI of c with B;

    $cost$ = predict the cost value of c with b;

    if $ei/cost > best\_value$ **then**

        $best\_configuration$ = c;

        $best\_value$ = $ei/cost$;

    end

end

return $best\_configuration$;

Algorithm 1: Hydra Hybrid variant algorithm.

3.2.3 Cost of identifying the next configuration to be evaluated

Hyperparameter optimization of large machine learning models can have a high economic cost [42], especially in scenarios where there is a need to rent computational power. In situations such as this, we want the optimization to be as efficient as possible so that the optimization economic cost is as low as possible and produces the best result. Unfortunately, without querying the model for all possible configurations, e.g., using a grid-based approach, one cannot guarantee to have correctly identified the configuration that the model predicts to be the optimum. In BOHB, there are always a fraction of configurations that are randomly sampled, and in scenarios like we have described previously where there are associated economic costs to each configuration that is sampled, having an under performing result can have an even more negative impact. These situations can't be avoided in Hydra as well since we need initial results to build a model, and we need Hyperband's theoretical properties. However, we can greatly reduce the economic cost of the optimization when using the model to predict a configuration. This is done by replicating the Gaussian Process model we use to predict the accuracy/loss of a given configuration, but instead of feeding performance-related information, we use economic-cost related information and finally, when calculating the Expected Improvement of a configuration, we divide it by the predicted economic cost given by this new model. When the search space is too big to compute exhaustively the acquisition function on all configurations, Hydra supports a simple heuristic that was already used in Fabolas [36], namely a mix of uniform random sampling and sampling via a gaussian

centered on the currrent incumbent. As an alternative, one could have used other black-box optimizers such as Direct [43] or CMAES [44].

Summary

In this chapter we have reviewed Hyperband and BOHB algorithms and explained how they would relate with Hydra. We have also described how Hydra would sample configurations and both analyzed and explained the thought process of the multiple variants included in Hydra, namely full-budget sampling, current-budgets sampling, expected improvement per dollar, and hybrid variations. In the next chapter we will evaluate Hydra in different environments and compare with related algorithms to point out what are its main advantages and disadvantages measuring loss, economic cost ($), Time and Overhead.

Chapter 4

Evaluation

This chapter evaluates Hydra via five different experiments. We first present the settings of each experiment. Then, we compare how each variant performs in each different environment measuring the loss of configurations achieved and accumulated cost ($) spent performing the optimization and the time taken. After this, we will select two of the best performing variants and compare them in the same experiments against state of the art algorithms that are related to Hydra, namely Hyperband and BOHB. We chose those algorithms because we want to establish experimentally if Hydra can outperform them in different scenarios. We include among the baselines also a variant of BOHB, which, instead of using the Tree-structured Parzen estimator, relies on the same modeling techniques used in Hydra. This allows for discriminating the effects of using different modeling techniques (TPE vs Hydra's GP-based acquisition functions) and of different input data sets (including or not the budget in the set of features fed to the models).

4.1 Test Environment

In this chapter we present five different experiments. Three of the five experiments only have a single difference, which is the machine learning model used to train the MNIST [45] dataset. This dataset is composed of 70000 28x28 pixel images of size-normalized handwritten digits from zero to nine and has training set of 60000 and a test set of 10000 image examples. These three experiments have the budget associated to the dataset size (number of images) used to train different neural networks. The minimum budget used was 3750 images, and the maximum was 60000 images. These three experiments were conducted using different neural networks, namely CNN [46] (convolutional neural network), RNN (recurrent neural network) [47] and a multilayer neural network [48]. CNN is a neural network that is mainly used in image recognition area, and is composed by two convolution layers, one input and output layer. The main idea how CNN extracts features from an image is by extracting small features from the previous layer like a feed-forward neural network and by applying pooling and convolution operations, these features will be given as input to the subsequent layers. plus two fully connected hidden layers with 256 neurons. These neural network models were trained in a public cloud, namely Amazon Web

Services (AWS), over a large number of configurations (288) and the corresponding cost and execution time were made publicly available [6] [7]. Table 4.2 has information about the hyperparameters used in these datasets. The configuration space considered in these experiments includes both parameters describing the type and amount of virtual machines to provision from the cloud and three models' hyper-parameters, namely learning rate, batch size and synchronization type. From here on we will mention theses experiments as CNN, RNN and Multilayer.

In CNN, RNN and Multilayer experiments we use as the economic cost measure the cost in dollars ($) of training a model with a given hyper-parameter configuration in AWS using the picked virtual machine type (which has an associated cost per second). Some other parameters of the optimization process in these experiments are equal, such as the intermediate budget values, as detailed in table 4.5, which are: 7500, 15000 and 30000 images. This combined with the minimum and maximum budget we declare for our experiment we can calculate what are resulting intermediate budgets that are available to the experiment. Encapsulating this list with the minimum and maximum budgets we get: 3750, 7500, 15000, 30000 and 60000. These will be the budgets used by the optimizer. It is also worth mentioning that RNN and Multilayer experiments are faster to train than CNN, given that CNN is much more complex; ergo, providing less economic costs in the training process.

CNN, RNN and Multilayer Experiment Search Space	
Hyperparameter	Values
Batch Size	[16, 256]
Learning Rate	[0.00001, 0.0001, 0.001]
Number of Workers	[8, 16, 32, 48, 64, 80]
Synchronization type	[asynchronous, synchronous]
Virtual machine Flavor	[t2.small, t2.medium, t2.xlarge, t2.2xlarge]

Table 4.1: Description of CNN, RNN and Multilayer hyperparameter values.

UNet Experiment Search Space	
Hyperparameter	Values
Machine type	[GTX 1080TI, RTX 2080TI]
Learning Rate	[0.000001, 0.00001, 0.0001]
Number of GPUs	[1, 2]
Synchronization type	[asynchronous, synchronous]
Batch Size	[1, 2]
Momentum	[0.9, 0.95, 0.99]

Table 4.2: Description of UNet experiment hyperparameter values.

machine type, batch size, learning rate, momentum, and synchronization type. The fourth experiment was selected as it was previously used in the evaluation of BOHB. As such this experiment focuses solely on the problem of hyper-parameter optimization, i.e., it does not include the type/amount of cloud resources in the configuration space. Analogously to the previous experiments, it also uses MNIST and a CNN. However, it considers a set of seven hyper-parameters, see table 4.3, yielding a total of 135000 possible different combinations. Unlike the previous experiments, where cost/execution times had been exhaustively measured and pre-recorded, this is not possible given the vastness of the search space. So, in the experiments reported below, Hydra is deploying and executing ML jobs using Keras

deep learning library and reading the resulting accuracy/loss (whereas in the previous experiments this process could be simply emulated by reading from the pre-recorded log file the configurations' cost and quality). This experiment has as optimizer a Stochastic Gradient Descent and only 8192 images are used for training, 1024 for validation.

CNN, RNN and Multilayer Experiment Search Space	
Hyperparameter	Values
Stochastic Gradient Descent Momentum	[0.0, 0.2, 0.4, 0.6, 0.8]
Learning Rate	[0.000001, 0.00001, 0.0001, 0.001, 0.01]
Number of Filter in Layer 1	[4, 8, 16, 32, 64]
Number of Filter in Layer 2	[0, 4, 8, 16, 32, 64]
Number of Filter in Layer 3	[0, 4, 8, 16, 32, 64]
Number of Hidden Units in the fully connected layer	[0, 4, 8, 16, 32, 64]
Dropout Rate	[0.0, 0.2, 0.4, 0.6, 0.8]

Table 4.3: MNIST hyperparameter values.

Given that exhaustively evaluating the acquisition function on all possible configurations is infeasible in this case, given the vastness of the configuration space, we only compute the acquisition function for 8000 configurations selected at random, where 70% are uniformly distributed throughout the whole search space and 30% sampled by centering a Gaussian on the current incumbent. We will be referencing this experiment as MNIST in the future. As cost metric we use time here, since a single machine was used in this experiment.

The final experiment involved a UNet [49] neural network adaptation in the context of Satellite Image Segmentation. This network uses Feature Pyramid Network [50] that has a size of 256*256*512 neurons with 1.2 gigabytes of training data. The loss function is soft-max cross-entropy, and the hyperparameters were machine type, batch size, learning rate, momentum, and synchronization type. In this experiment we performed the training in GPUs and used two different machines, one with two GTX 1080Ti, and another with two RTX 2080Ti, and measured economic cost as if they were rented in AWS. Since AWS does not possess instances with these GPUs, we use as pricing the instances with the nearest ratio of floating-point operations every second on average, which are Tesla K80 (P2.xlarge) that cost 0.9$/hour and will correspond to the machine with GTX 1080Ti, and Tesla M60 (G3.4xlarge) that cost 1.14$/hour that will correspond to the machine with RTX 2080Ti. These costs need to be multiplied by the number of GPUs used in each configuration. In this experiment we have budget as wall-clock time and we have the maximum budget as 5 hours and the minimum budget as 18 minutes and 45 seconds.

In the following section we will explain the workflow of the optimizer in our experiments in order to clarify the analysis of the results.

4.2 Experiment Run-through

In all of the experiments, we complete, for each run, 10 iterations of each optimization algorithm — recall that all the optimization algorithms are based on the Hyperband base scheme. Each iteration, called

also ”bracket”, is composed by a number of ”stages” that is equal at most to the amount of different budgets used by the optimizer, which in out scenario is 5.

Each stage will contain a number of different configurations that will be evaluated (e.g., by training a neural network and observing the resulting accuracy) and each stage will have an associated budget. Each stage evaluates a pre-determined number of configurations on the budget associated with the stage, and then run the Successive Halving algorithm, where the top ranked configurations (with respect to, i.e. lowest loss value) will pass to the next stage. The next stage will have the nearest greater budget in the budget list used by the optimizer. This will repeat until the stage budget is equal to the maximum budget value.

When a bracket/iteration is over, a new one is activated using the next larger level for the initial budget. This process repeats until a bracket is activated with the starting budget as the maximum budget. Next, if the optimizer has not finished all its iterations, the next bracket will have as starting budget the minimum budget value. Thus, in our settings, we execute two full cycles of the Hyper-band base scheme.

SH#	Initial budgets				
	3750	7500	15000	30000	60000
	Bracket 1	Bracket 2	Bracket 3	Bracket 4	Bracket 5
0	16	8	4	4	5
1	8	4	2	2	
2	4	2	1		
3	2	1			
4	1				

Table 4.4: Hyperband bracket decomposition with maximum budget = 60000, minimum budget = 3750.

The number of configurations to sample in each bracket are determined by its starting budget value, and as per the Hyperband algorithm, the total number of generated configurations for each bracket is always greater than the number of different possible brackets that the algorithm can generate. For instance, as we can see in Table 4.5, in CNN, RNN and Multilayer experiment we generate 5 different brackets, and each bracket will have 16, 8, 4 , 4 and 5 initial configurations and start as initial budgets 3750, 7500, 15000, 30000 and 60000 respectively. Thus by executing Successive Halving the first, second and third bracket, the optimizer will execute one configuration in the maximum budget in each bracket, whereas in the fourth and fifth bracket it will produce two and five configurations on the maximum budget value respectively.

To recapitulate, the first iteration will form the first bracket that will have the minimum budget value and starting budget. So, in the first stage of the first bracket it will sample 16 different configurations and evaluate them (sequentially) with respect to the objective function with the minimum budget value.

After evaluating all of the assigned configurations, it will pass the top configurations to the next stage. The next stages starts with the top configurations of the previous stages and evaluates them with the immediately larger budget value (which, for CNN, RNN or Multilayer experiment will be 7500) and like the the previous stage, will rank and pass the top (best-performing) configurations to the next stage. After the last stage with the maximum budget value evaluates the last remaining configuration, the bracket will finish, and a new iteration will begin, forming a new bracket that will have 7500 as starting budget

value (in CNN, RNN or Multilayer experiment). In Successive Halving, in this experiment with the given parameters we use top half of the evaluated configurations since this is corresponds to the inverse of the eta value (Hyperband algorithm details). In total, 10 iterations will generate 128 evaluations of configurations. In the next section we will discuss some information about the plots and how to visualize them.

4.2.1 Plotting details

To simplify the visualization of the plots, after the first bracket (31 explorations) we start plotting the incumbent of each optimizer. This is in fact the first point in which a full budget configuration is evaluated, thus allowing to establish the notion of currently known optimum.

When an optimizer fails to find a better incumbent, we plot a black circle with the current incumbent loss, however when it upgrades an incumbent we plot the point similarly to the points in the first bracket. For each plot we mark with a red square the best achieved loss value for each optimizer. We also plot the variance of the plotted values with the same color as the optimizer. The results are based in 300 runs of each optimizer (10 iterations per run) for the CNN, RNN and Multilayer experiment, 50 for UNet dataset and 20 for MNIST dataset. All the runs use deterministic seeds to initialize the random number generators in each different run, making it possible to replicate the same results for every different run. In the next section we will start by presenting the results of every experiment with respect to the performance of the various variants of Hydra that were developed in this dissertation.

4.3 Hydra variants comparison

In this section we will start by presenting the experiment results with respect to the wall-clock time, analyzing the most relevant plots and pointing out interesting behaviors observable in our experimental data. Next, we shift perspective to consider the economic cost of each experiment. Finally, we present conclusion that sum up what are the pros and cons of each variant and what can be considered the best performing one on different levels.

4.3.1 Duration of the optimization process

In this experiment we can effectively measure how fast is each variant with respect to each other, what are the advantages of having a less complex system with a single model and what can cost-reducing variants can do. Since we have several experiments that have different environments, we will group the more similar ones and evaluate them together.

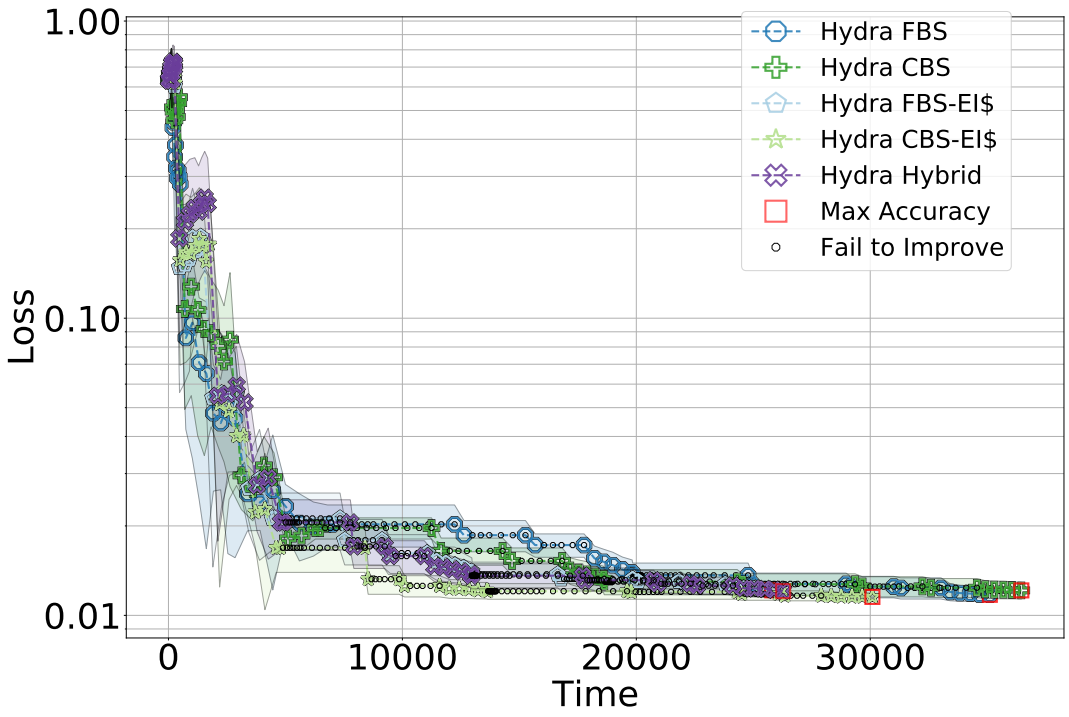


Figure 4.1: Total time (seconds) and Loss (in log scale) in CNN experiment

CNN, RNN and Multilayer We can note in Figures 4.1 and 4.2, FBS and CBS manage to get slightly better configurations in the early stages of the algorithm, especially FBS, showing that the configurations that it has picked have the potential to be better than the other variants. However, when the last sample of the first iterations finishes, CBS with EI per $ manages to achieve the best configuration of the first iteration. After some time passes, we can see that CBS with EI per $ maintains its superior pace, always achieving a higher accuracy than the rest. We can also notice that FBS with EI per $ and Hybrid finish all ten iterations, albeit with a higher loss than CBS with EI per $, very early, while CBS and FBS finish almost 3 hours later. In these experiments, especially in CNN, higher wall-clock times are closely related with high cost, since these experiments are ran in a cloud computing platform, where time is equal to money, so by consequence, cost-reducing variants are likely too have lower execution times (at least when run on the same type/amount of cloud resources). FBS with EI per $ is the variant that finishes first in CNN (lowest execution time), this is many because given that sampling a configuration in a larger budget will produce higher costs. This is true for these experiments because cheap configurations take on average 50% less time to complete. As explained in section 3.2.1 it would have a greater affect the higher the cost, and it so happens that this CNN experiment has a higher cost than RNN and Multilayer. Contrarily, it is expected that FBS and CBS to be the slower variants, since they do not care about time or economic costs.

What is surprising however is the performance on CBS with EI per $. In RNN, where the costs are significantly lower (since the sample time in each network is much lower), we can see a degradation

of performance on CBS with EI per $. However, in both RNN and Multilayer, FBS with EI per $ and Hybrid prove to be consistently faster than other variants, finishing the experiment with 50% lower time on average than variants with no economic-cost reduction. We can see the affect this has especially on FBS in RNN, where it seems to lag behind and sample configurations in a much slower pace than the rest.

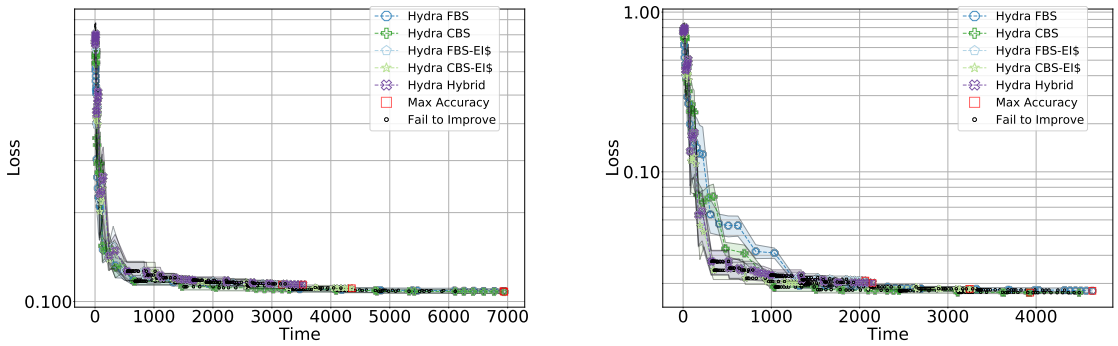


Figure 4.2: Total time and Loss in log scale in Multilayer experiment (a) and Total time and Loss in log scale in RNN experiment (b)

MNIST Since this experiment does not possess any type of economic cost related value, we have set the economic cost equal to the time spent for the variants (e.g., the ones that use EI/$) that keep cost into account. Since the used budget for this experiment is epochs, the higher the budget, the longer the time spent in the training process. We can see that Hybrid variant is the one that finished first the ten iterations, having 18% lower execution time on average than CBS. In spite of being slow, CBS has on average the best incumbent values, proving that by sampling towards bracket budget values will get more consistent results. However FBS achieves on average the highest accuracy on all points than the rest of the variants and it also surpasses CBS incumbent values at a later stage, which shows the strength of sampling towards the full budget value will show the best results on later stages.

UNET In this experiment, analyzing loss values against time done in figure 4.4 , we can see that the explorations are synchronized, that is, all variants transition to the same stage and brackets simultaneously. This is because the budget type in this experiment is wall-clock time, so it is expected that they finish each bracket at the same amount of time. Contrarily to previous experiments in this section, we cannot evaluate the difference in execution time. We can see in figure 4.4 (a) a difference in loss values obtained in the different iterations. We can also see that FBS variant does not perform as well as other experiments in the early stage of the optimization, being equally matched with CBS variant in the loss values obtained. In the subsequent iterations in (b) we see that FBS does not achieve lower loss values than CBS, even in the last iteration.

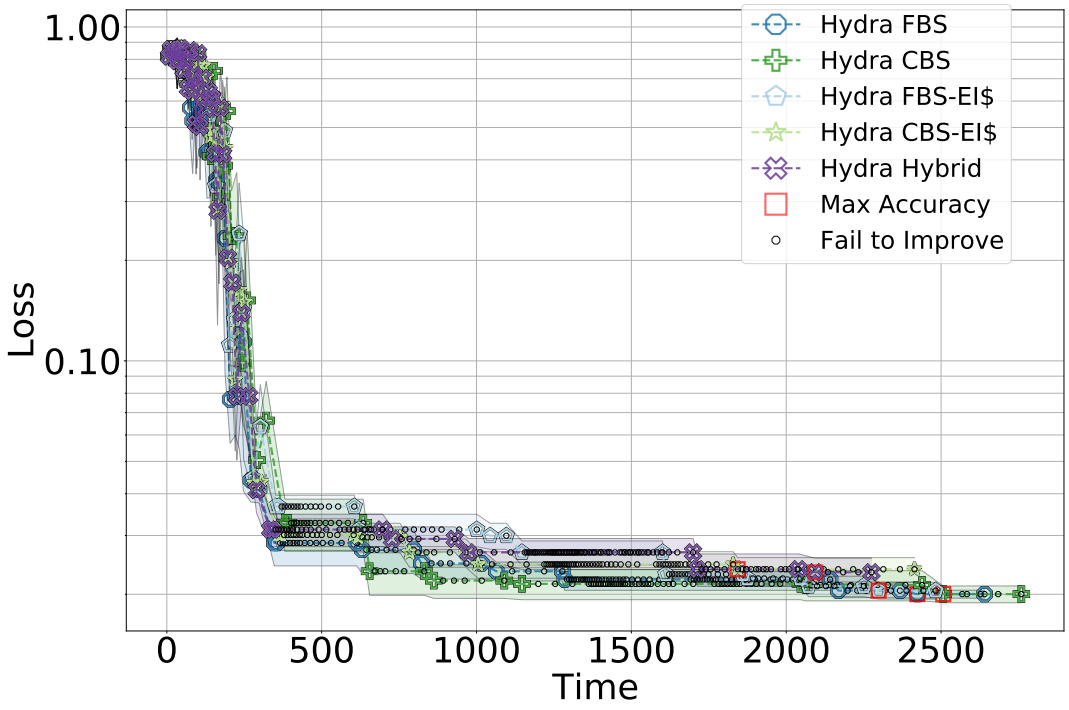
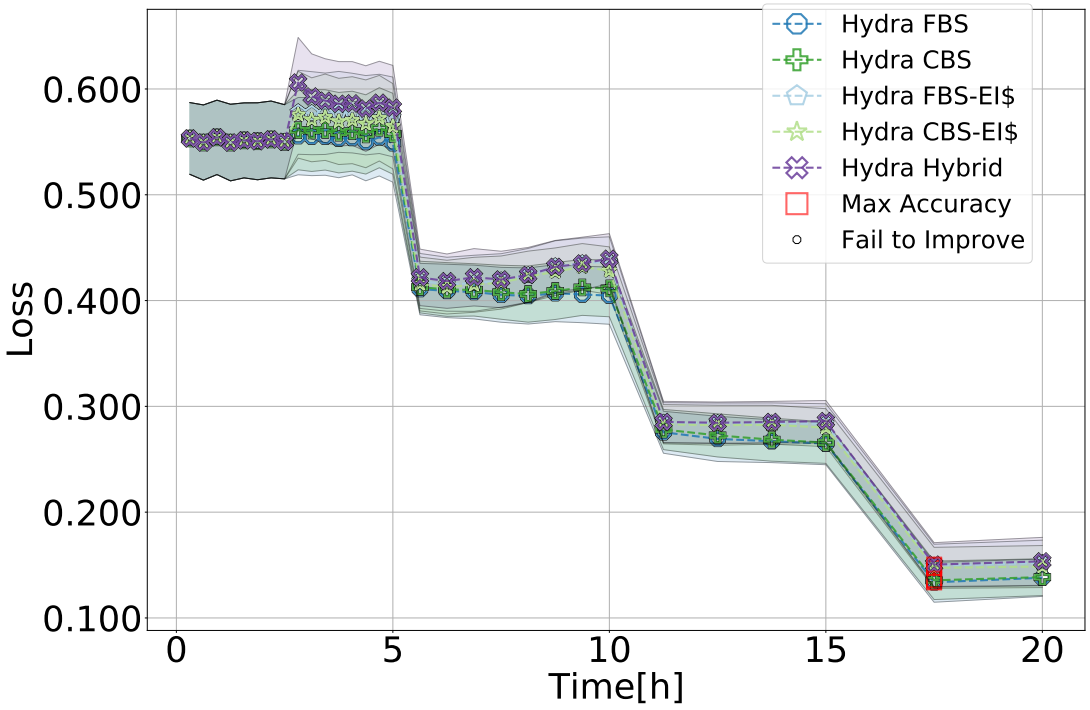


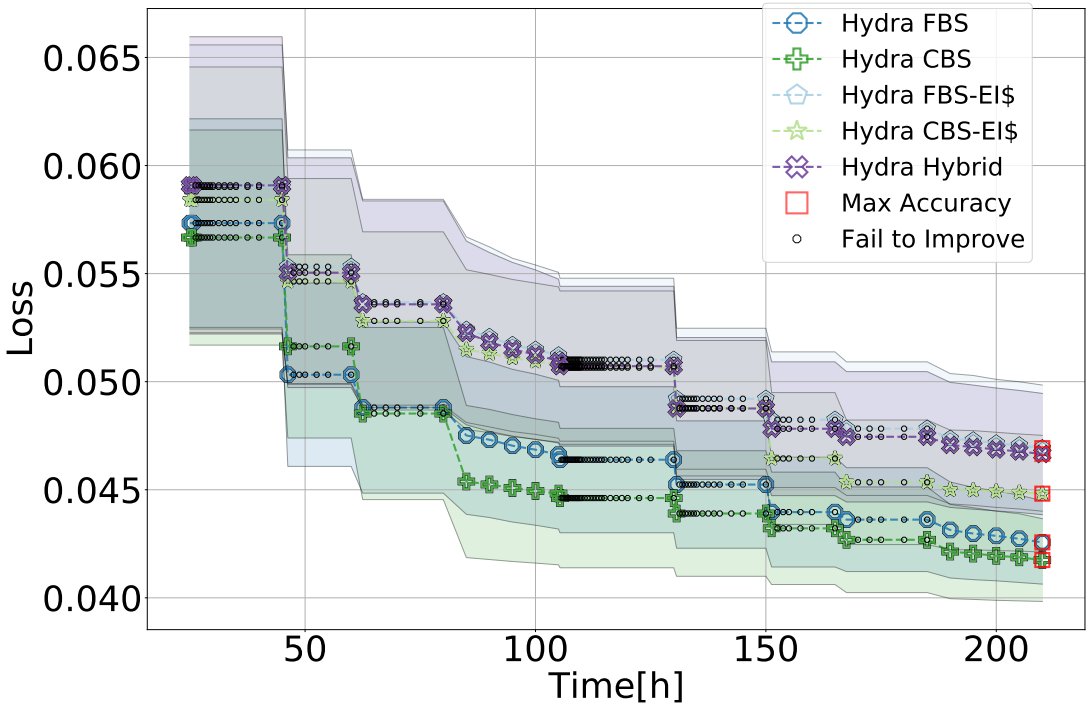
Figure 4.3: Total time (seconds) and Loss in log scale in MNIST experiment

4.3.2 Cost of the optimization process

This experiment analyzes the economic cost of the optimization process for all variants of Hydra. Like section 4.3.1 since we have several experiments that have different environments, we will group the more similar and evaluate them together. **CNN, RNN, Multilayer** In figure 4.5 (a) we can observe a dominant gain of FBS in the early stages of the algorithm, being able to identify configurations that are generally better than the rest of the variants. However other variants catch up quickly, as can be seen in Figure 4.5 (b). Further the variants that incorporate a cost model in their acquisition functions all attain higher quality configurations at a lower cost. This is particularly clear for CBS with EI per $, which achieves the smallest lost at around half the costs of the FBS and CBS variants. Similar considerations apply to RNN and Multilayer, in Figure 4.7 (a) and (b) respectively: also in these experiments, CBS and FBS with EI per $, as well Hybrid, achieve approximately a 2$\times$ cost reduction, albeit in this case at the expense of a slight degradation in the accuracy of the configurations recommended at the end of the optimization process.

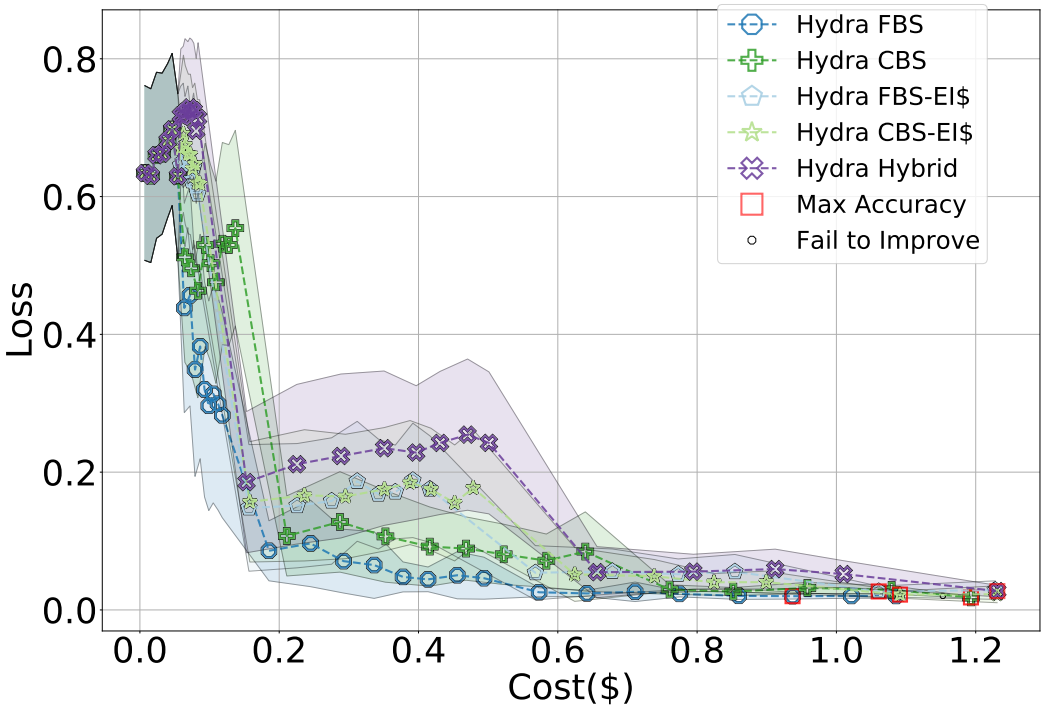


(a)

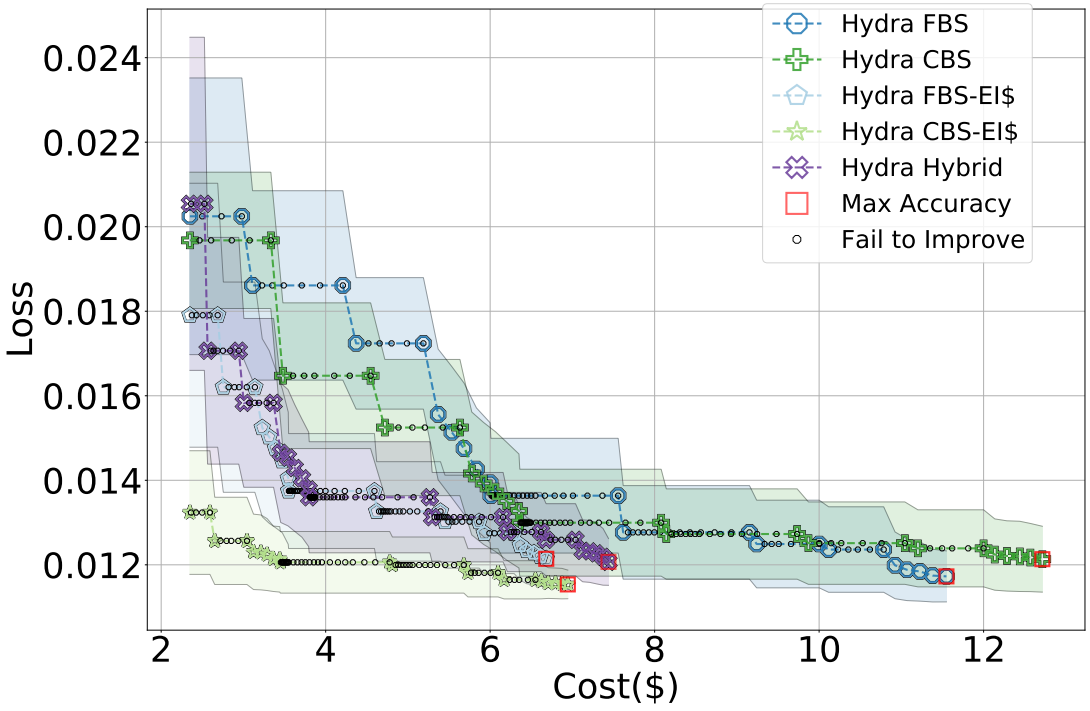


(b)

Figure 4.4: Total time and Loss in UNET experiment focused in 1st Iteration (a) and focused on the remaining Iterations (b)



(a)



(b)

Figure 4.5: Accumulated cost (dollar) and loss in CNN experiment scaled in the 1st iteration (a) and then scaled in the remaining iterations.

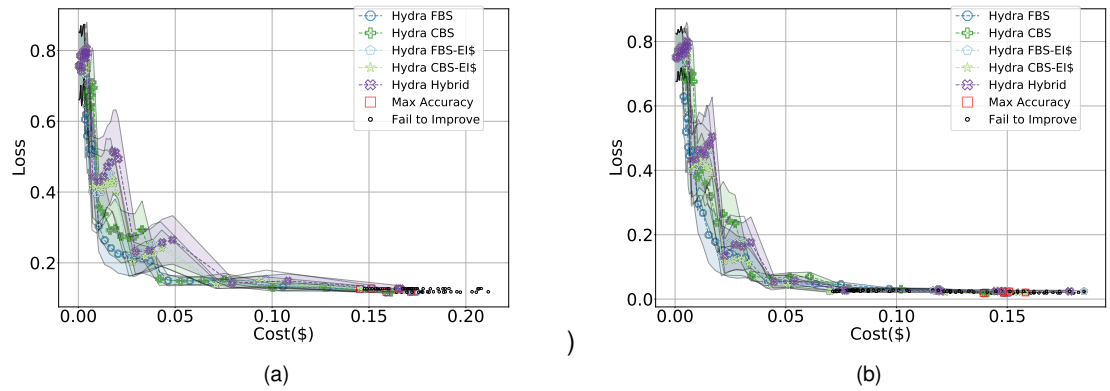


Figure 4.6: Total time and Loss scaled in the first iteration in Multilayer experiment (a) and in RNN experiment (b)

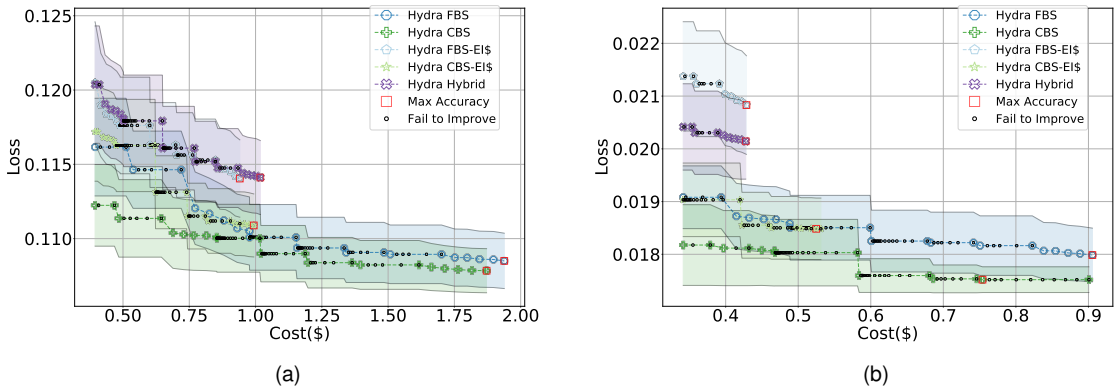
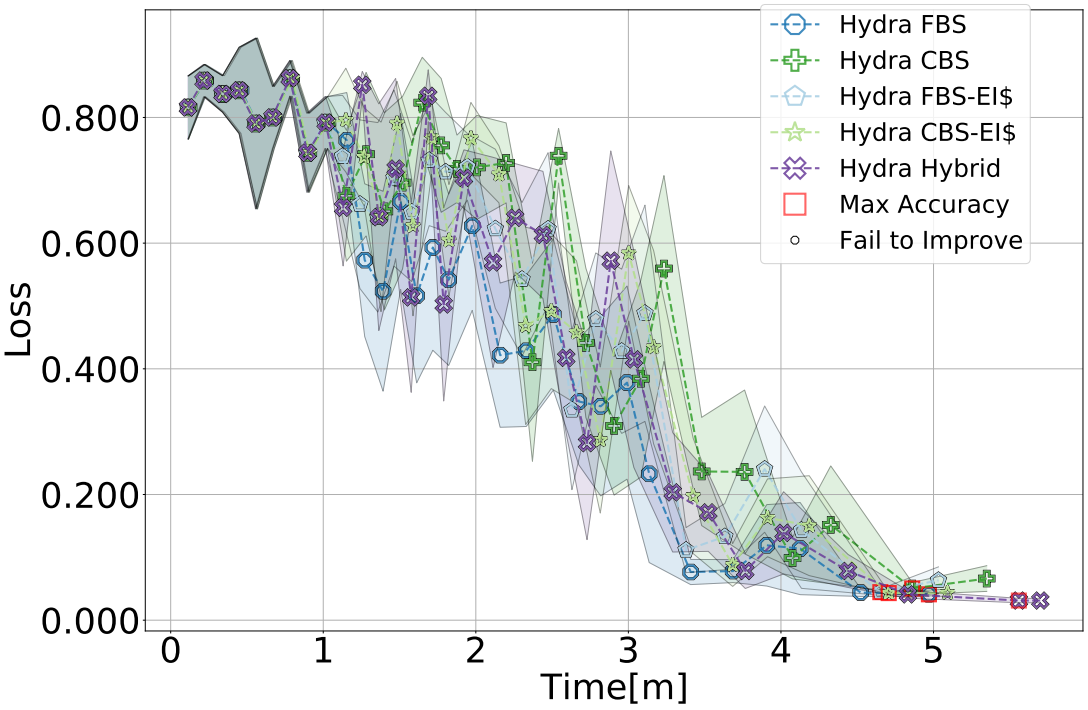
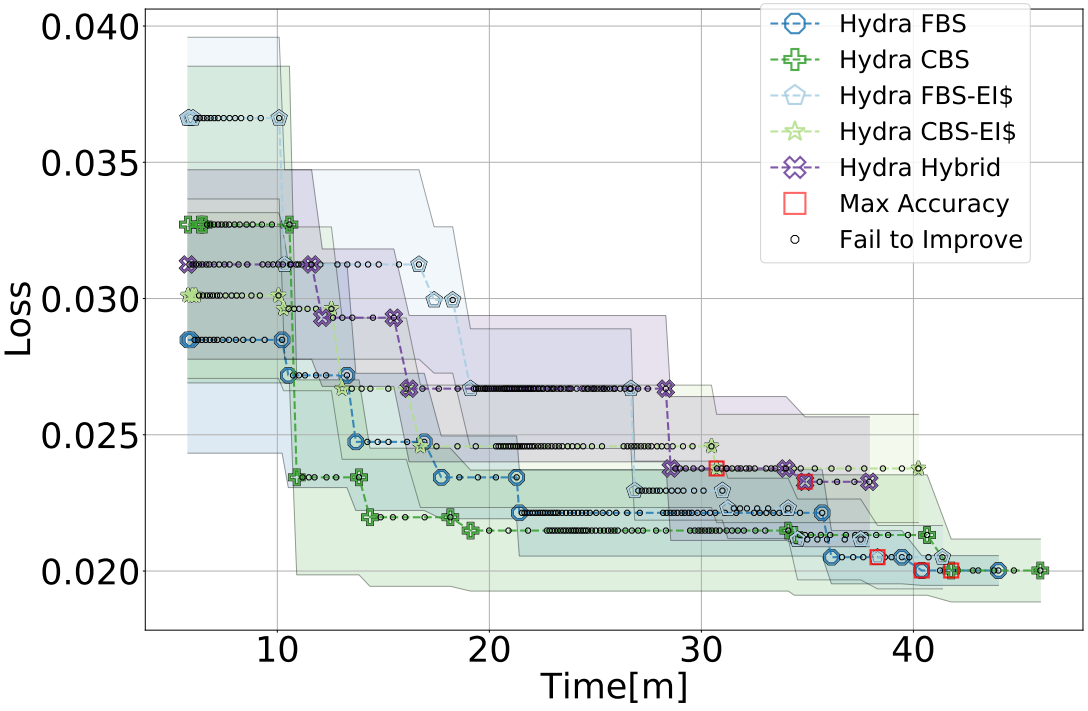


Figure 4.7: Total time and Loss scaled in iterations 2 to 10 in Multilayer experiment (a) and in RNN experiment (b)

MNIST here we do not show cost-related figures, since as previously discussed, this experiment lacks a valuable source of economic cost, (it is equal to time). This will hamstring economic-cost focused variants, resulting in lower cost reductions and average below performances with respect to FBS and CBS. However, we introduce here figure 4.8, that has been scaled to perform a closer visualization with x as wall-clock time with minutes. We can see an advantage of FBS in the early stage of the optimization process (a), noticed in the previous experiment, is less noticed, but still observable. This advantage falls off like the other experiments, and in figure 4.8 (b) we can see that CBS will maintain the best incumbent until FBS finds a better incumbent in the final iterations.



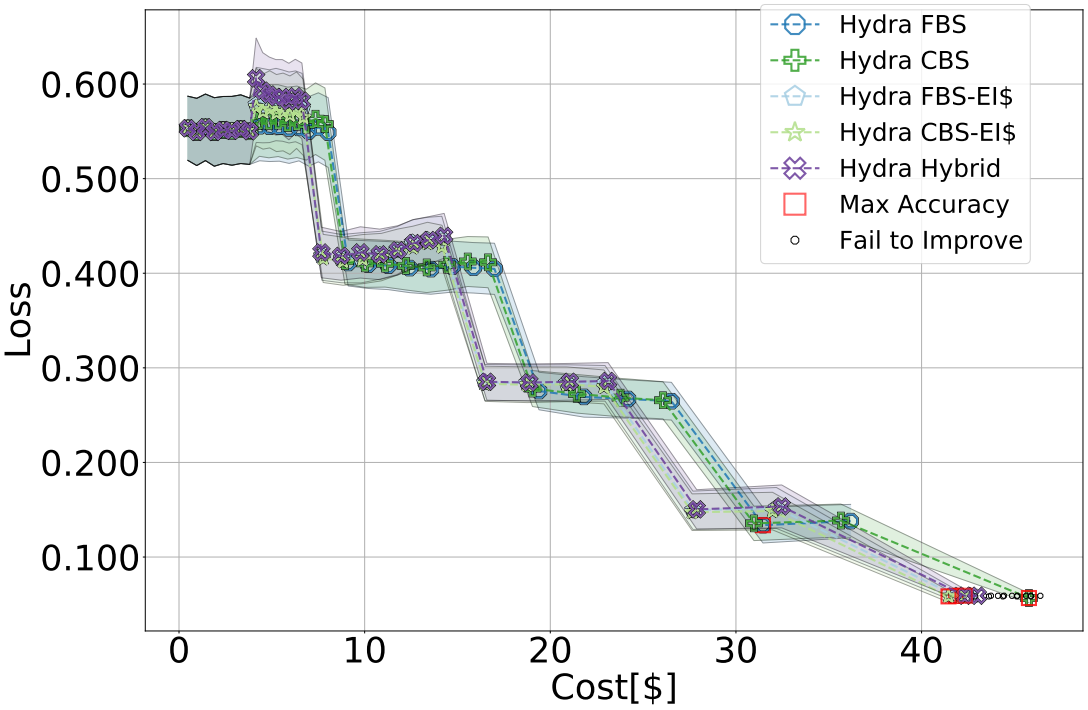
(a)



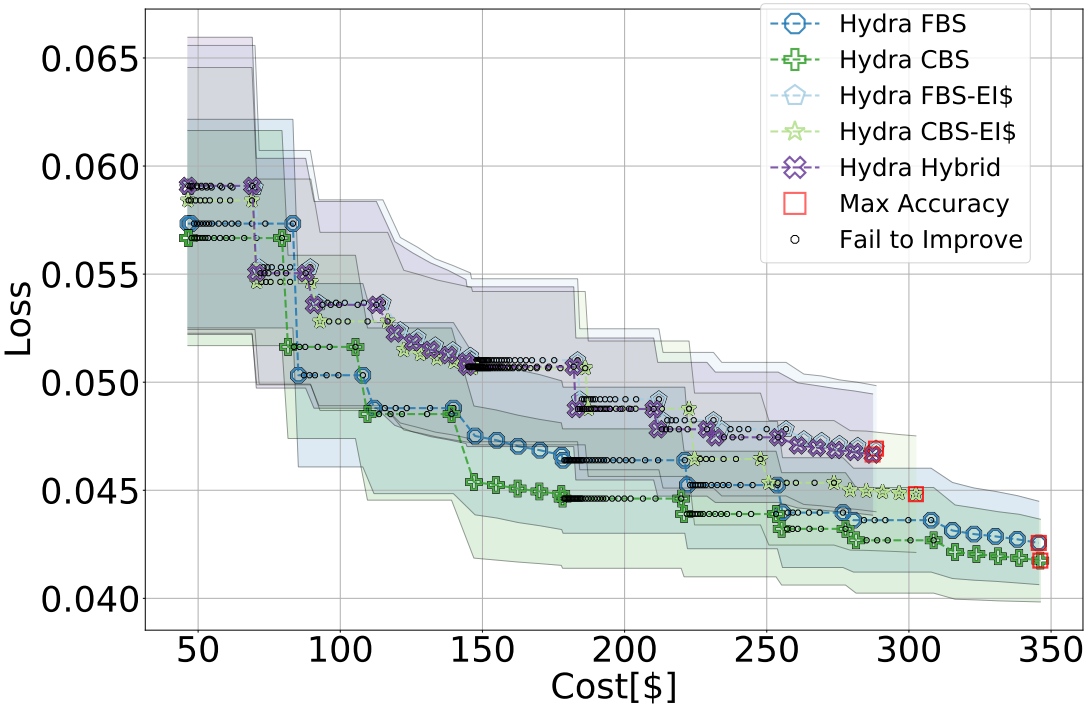
(b)

Figure 4.8: Accumulated wall-clock time (in minutes) in MNIST experiment focused in 1st Iteration (a) and focused on the remaining Iterations (b)

UNET comparing figure 4.9 (a) with 4.4 (a), we can notice a difference in performance of economic-cost variant performance, where with lower costs, they can transition to other stages, being able to achieve lower loss values. In (b) however, it is apparent that by being cost efficient, they lack the performance of FBS and CBS variants. In spite of this, FBS with EI per $ and Hybrid variants will be on average 60$ cheaper (corresponds to 17.1% less costs) than CBS or FBS, while spending the same amount of time training and have a increase of 11% on the best achieved loss value. FBS in (b), like in figure 4.4 (b), has the same behavior as CBS, and does not achieve better results.



(a)



(b)

Figure 4.9: Accumulated Cost in dollars [$] in UNET experiment focused in 1st Iteration (a) and focused on the remaining Iterations (b)

4.3.3 Summary

Evaluating the variants from wall-clock time demonstrated that economic-cost reducing variants have the ability to speed up the optimization time of the whole run, consistently showing that it is faster than methods that do ignore economic cost. We have also shown that in spite of being relatively slow with respect to variants that take into account the economic cost of an experiment, FBS has demonstrated in the majority of experiments that it is able to produce better results in the beginning of the experiment which is even more apparent in the zoomed-in plots in section 4.3.2. Even if its relative performance degrades before reaching the end of the first iteration, FBS is able to produce good results at later stages of the optimization process.

By comparing all economic cost related variants, we can say that FBS demonstrated to be the variant that is consistently better at sampling in earlier stages than the rest. However CBS with EI per $ proved to be both surprisingly fast and cost efficient. In the next section of evaluation, we will be looking at how these two variants (FBS and CBS with EI per $) compare with respect to various state of the art optimizers.

4.4 Comparison with state of the art optimizers

In this section we present the results gathered using the CNN, RNN and multilayer neural network datasets and aim at evaluate the performance of the FBS and CBS with EI per $ variants of Hydra against two state of the art optimizers, namely Hyperband (HB) and BOHB (BOHB-TPE). As previously mentioned, we have also included a BOHB variant that uses Expected Improvement as the acquisition function (and the same Gaussian Process models as in Hydra) to isolate the gains deriving from incorporating information on configurations using different budgets in the model (as Hydra does).

We will consider the same experiments as in section 4.3 and, also in this case, we start by analyzing the accumulated economic cost and loss. In addition, we will provide a table which will contain the optimization overhead and additional information about and finally will sum up the analysis.

4.4.1 Cost of the optimization process

CNN In Figure 4.10 we can see the cost accumulation progression with respect to our elected variants of Hydra, namely FBS and CBS with EI per $, when compared with Hyperband, BOHB and BOHB with EI. This figure is focused on the first iteration, and we can clearly see that the FBS variant has a significant advantage over BOHB-EI, and CBS with EI per $ variant (as seen in section 4.3.2). This advantage is even more apparent with respect to BOHB-TPE and Hyperband. By analyzing, the remaining 9 iterations, see Figure 4.11, we see that the performance of FBS is closely matched with BOHB-EI. Both of them are shadowed by Hydra CBS with EI per $, which is able to reach better configurations with almost half the cost. Comparing CBS with EI per $ with BOHB-TPE, we can even see a clearer advantage, where with just only 2$ it is able to match its minimum loss value throughout the hole experiment, spending on average 88% less.

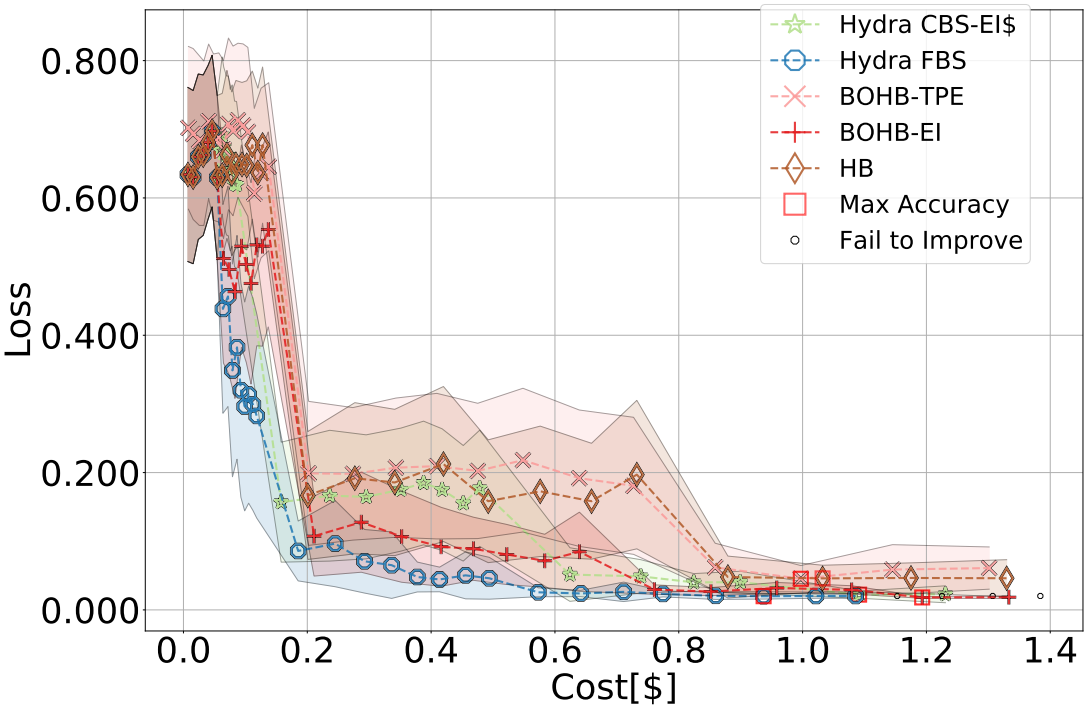


Figure 4.10: Accumulated Cost ($) and Loss in CNN, scaled in the first iteration.

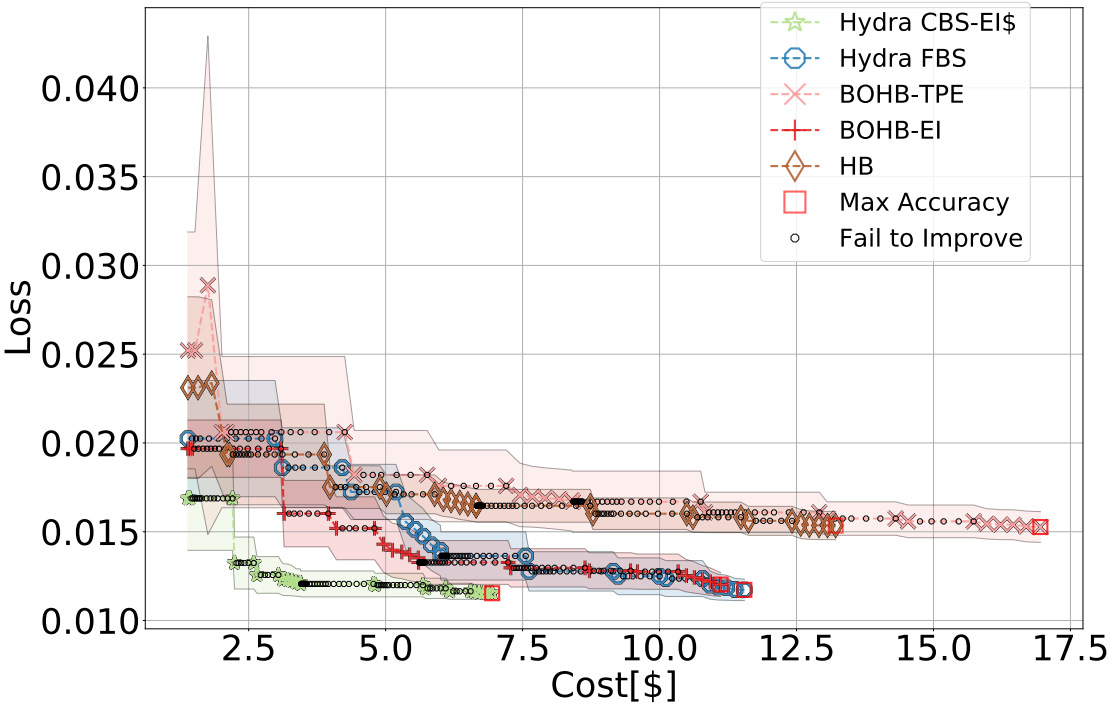


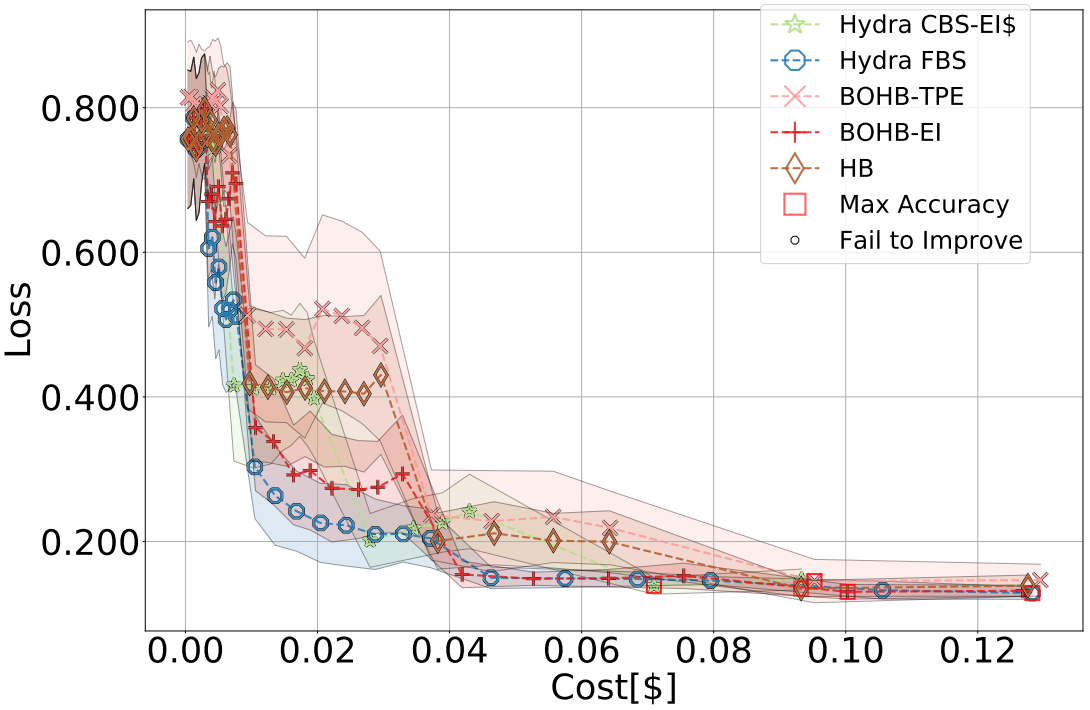
Figure 4.11: Accumulated Cost ($) and Loss in CNN, scaled in the 2nd to 10th iterations.

RNN and Multilayer for FBS and CBS with EI per $, the best performing variants would be by contrast CBS and CBS per $, as viewed in figure 4.7. We show the gains of using CBS in these experiment in figure 4.12 and 4.13. In these particular experiments, CBS is better than FBS and it also proves to be better than BOHB-EI. Interestingly CBS shows the exact same behavior as BOHB-EI in figure 4.12. This is because CBS samples always according to the same budget and it does not know any other result outside of the initial budget. Since the training set's configurations contain the same budget value the model wants to predict to, it will treat the configuration as if it does not have a budget hyperparameter, because it has no knowledge of any other configuration with a different value and it does not change the predicted value until it finishes the bracket. After all, if CBS ignores the budget dimension it essentially becomes like BOHB-EI in iteration 1.

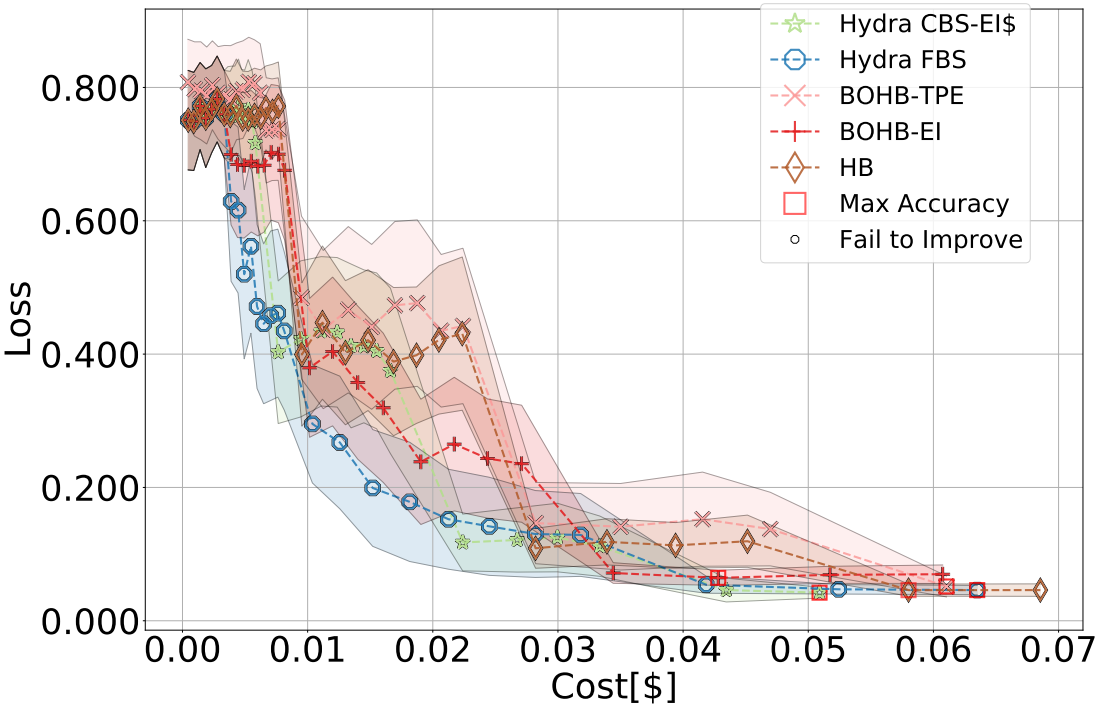
We can verify the advantage of inter-budget knowledge by the gains showed in figure 4.12. In these plots we can view a great advantage of using Hydra comparing it with Hyperband and BOHB-TPE, and even with BOHB-EI, which is indirectly a "enhanced" version of BOHB.

MNIST Since MNIST has its economic budget value equal to wall-clock time, we will show the comparison on section.

UNET In figure 4.14 (a) we can see that CBS with EI per $ manages to achieve better configurations with lower cost values than any other systems. It is similar to figure 4.9(a). We can see that in this experiment FBS does not distinguish itself from other variants, closely matching their loss values with the same cost. In figure 4.14 (b), we can see that FBS has in general better performance than other variants. Hyperband has great results too, which can indicate that this experiment is very hard to model. We can also see that CBS with EI per $ has in general, better performance than BOHB-EI and especially BOHB-TPE, and it also has lower costs. BOHB-TPE has on average the largest cost value.

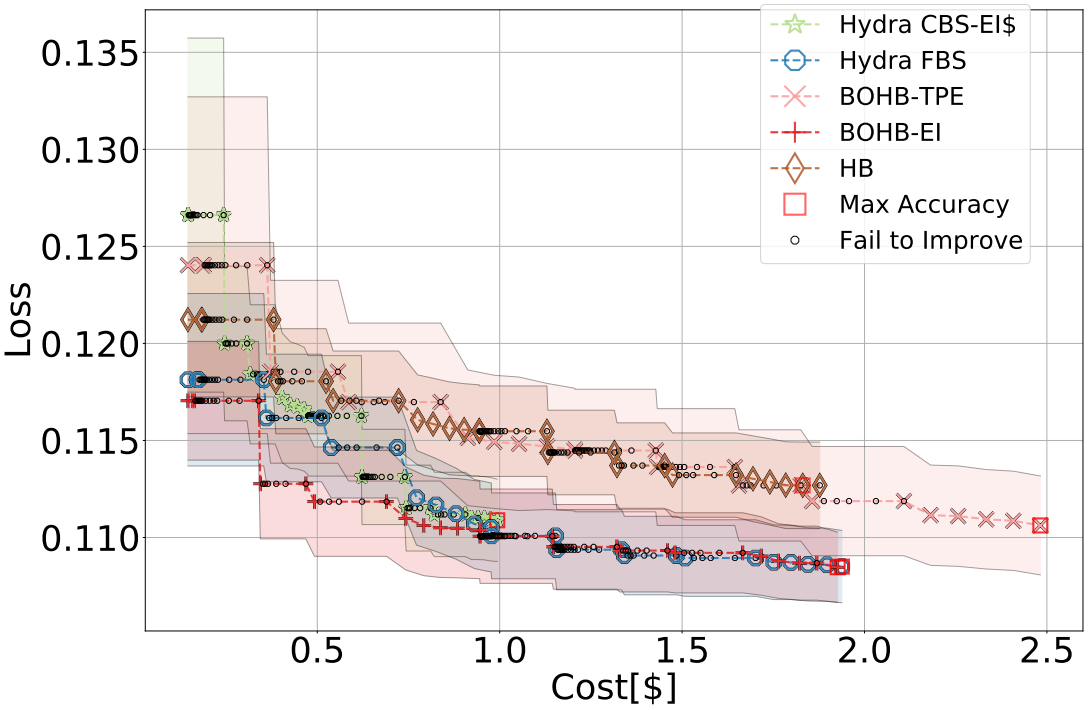


(a)

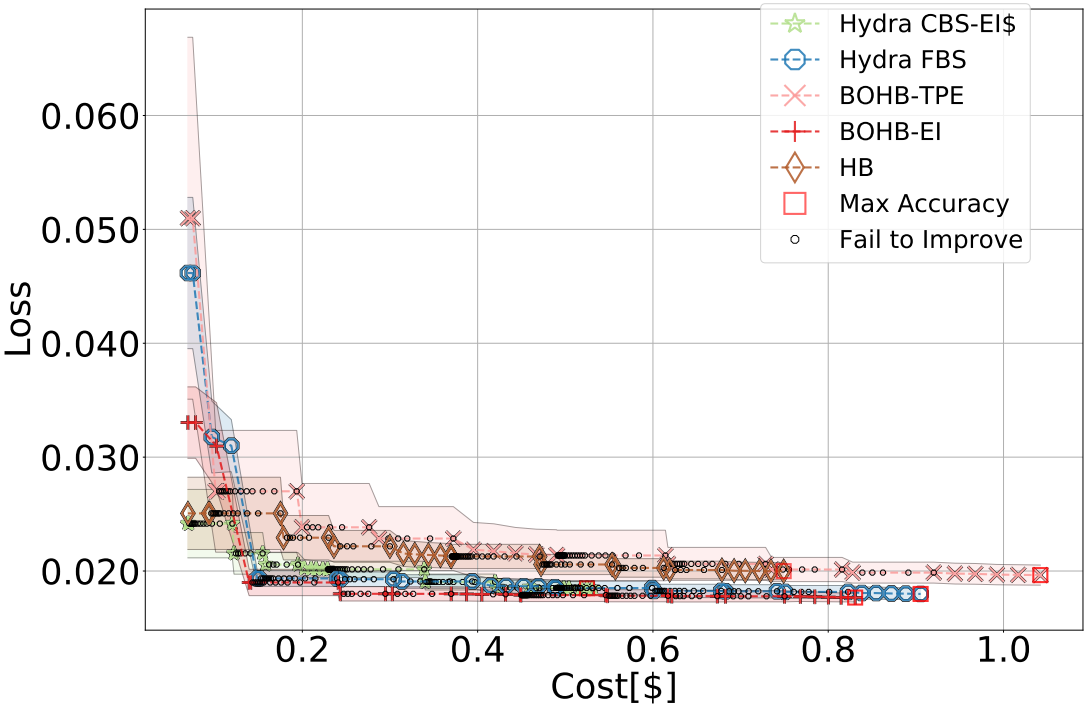


(b)

Figure 4.12: Accumulated and Loss with RNN (a) and Multilayer in log scale (b)

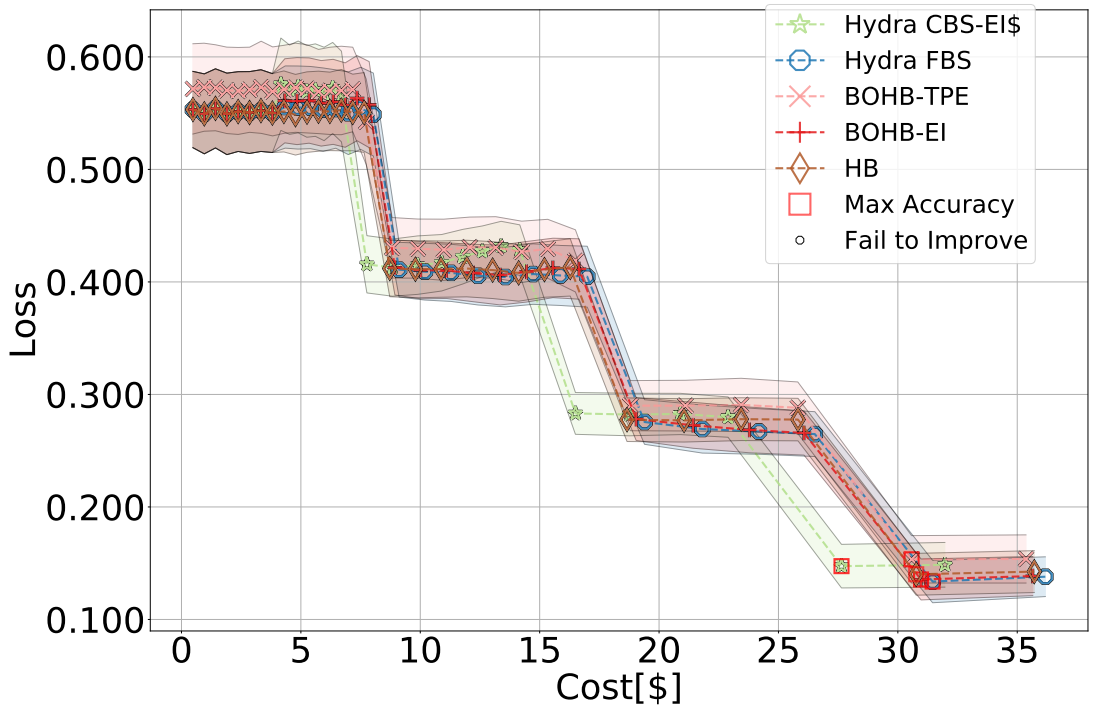


(a)

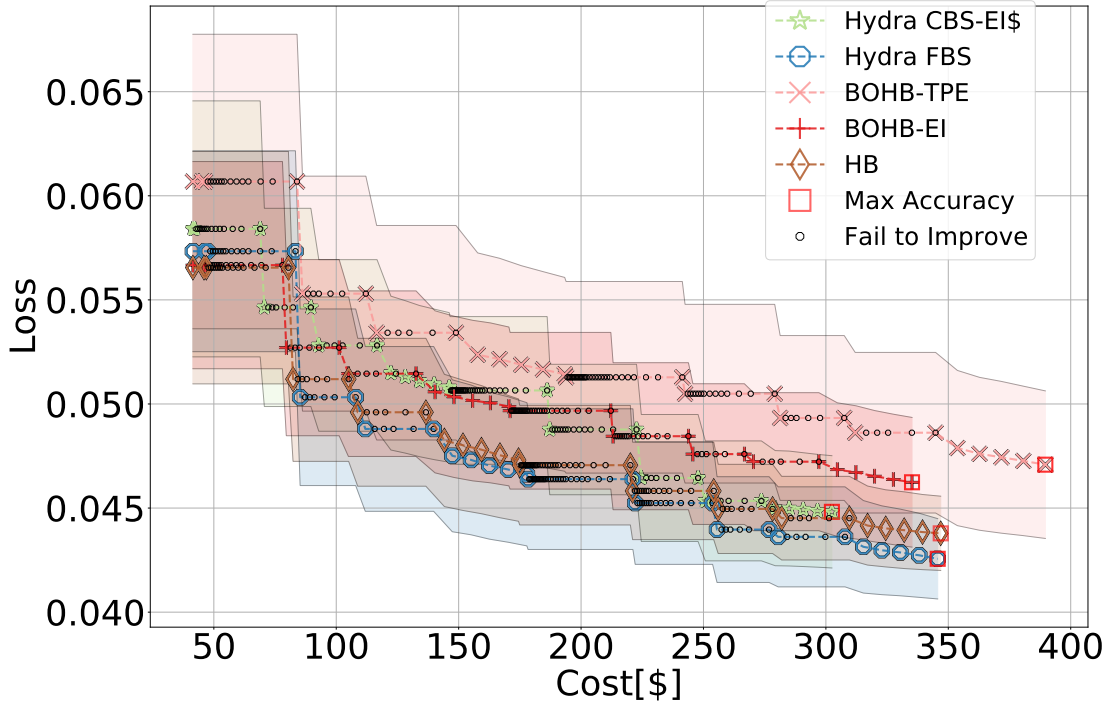


(b)

Figure 4.13: Accumulated and Loss with RNN (a) and Multilayer (b)



(a)



(b)

Figure 4.14: Accumulated Cost in dollars [$] in UNET experiment focused in 1st Iteration (a) and focused on the remaining Iterations (b)

4.4.2 Hydra Overhead

In this section we compare the overhead values obtained during the experiment. As demonstrated in table 4.5 we can view the accumulated overhead of each system in each experiment. Hyperband always has an almost non existing overhead and is because it always randomly samples the configurations that are to be evaluated. Across all experiments we see that BOHB-TPE and Hyperband have the lowest total overhead value, and especially in MNIST experiment, this difference is very noticeable. This happens because the Tree-structured Parzen Estimator used in BOHB-TPE is much faster than the Gaussian Process models used in Hydra, and this is exacerbated in UNET experiment because the search space is almost $500x$ larger than CNN, RNN and Multilayer experiment, and three orders of magnitude larger than UNET. This is minimized by only computing the EI of 8000 configurations in maximum, but it still has a sizable difference.

	Accumulated Overhead[s] over 258 explorations				
Experiment	Hydra FBS	Hydra CBS-EI/$	BOHB-EI	BOHB-TPE	Hyperband
CNN	21.5	26.9	17.6	5.4	0.2
RNN	20.3	26.3	16.4	5.3	0.1
Multilayer	20.0	25.8	16.8	5.4	0.1
MNIST	596.4	803.9	504.6	172.8	0.2
UNET	9.7	13.3	8.5	3.8	0.1

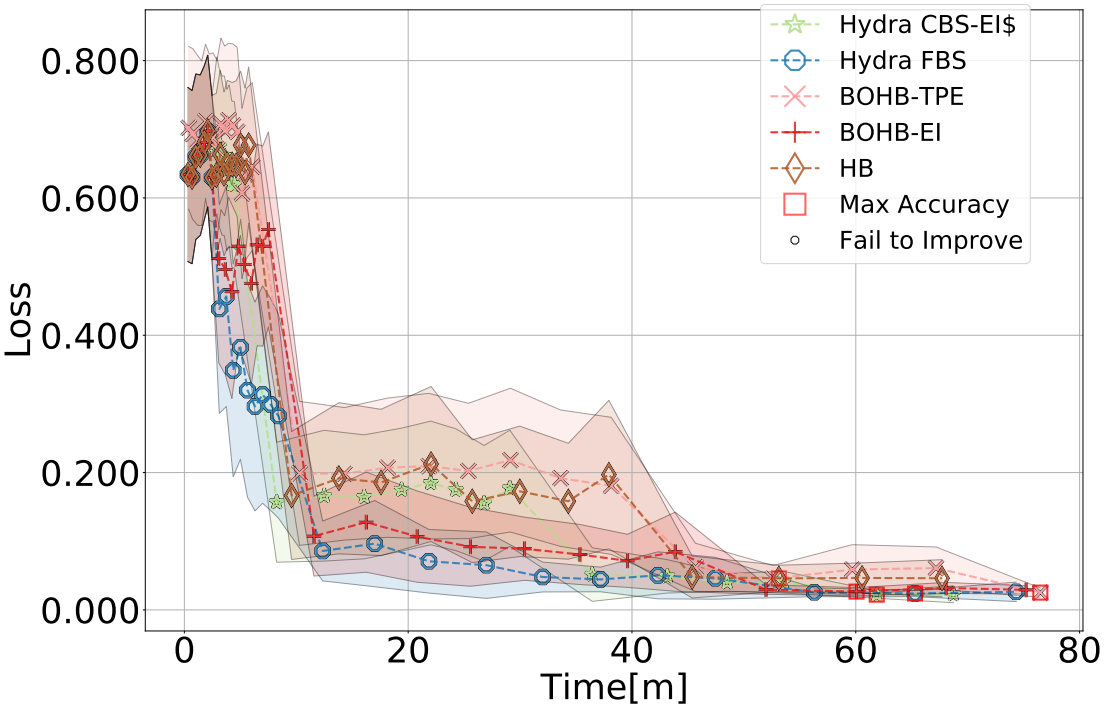
Table 4.5: Overhead value for each system in each experiment

4.4.3 Duration of the optimization process

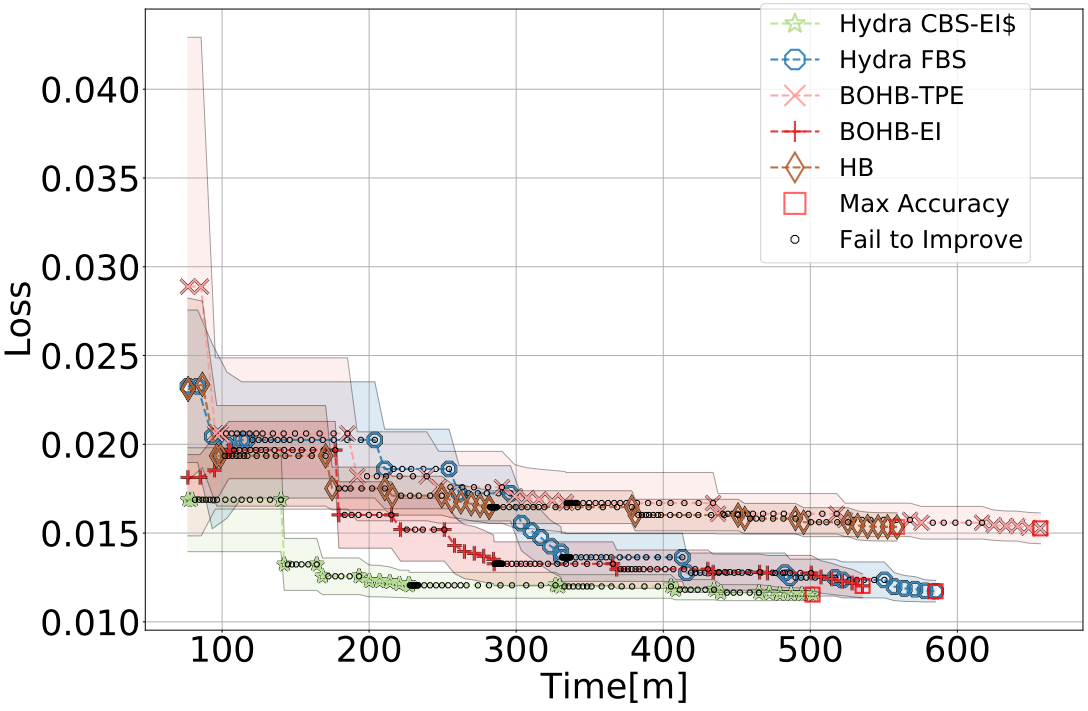
Here we take a look at the performance of the same selected optimizers but in a wall-clock time perspective.

In CNN we can identify in figure 4.15 (a) a major advantage in FBS, managing to quickly sample configurations that achieve a lower loss than other variants. This advantage is maintained until minute 60, where other optimizers manage to achieve similar values. Looking at 4.15 (b) we can see the predominance of CBS with EI per $, managing to achieve lower loss configurations than any other optimizer and finishing the optimization process earlier too. BOHB-EI matches Hydra FBS behavior, proving FBS only to be exceptional during early stages of optimization. Comparing Hydra variants to Hyperband and BOHB-TPE we find that they can at any moment achieve better configuration loss values in the same amout of time.

RNN and Multilayer in figure 4.16 show FBS prowess as well albeit in much shorter periods of time in (a) and (c). In (b) and (d), where iterations 2 to 10 are shown, we see a decrease of performance of CBS with EI per $ but we see that it has a much shorter period of execution and still manages to outperform Hyperband and BOHB-TPE. Like figure 4.15 (a), FBS has similar behavior as BOHB-EI, with the difference being that FBS lowers its incumbent loss values with a more visible pace, demonstrating an impact on intra-budget information gain.



(a)



(b)

Figure 4.15: Accumulated wall-clock time in minutes in CNN experiment focused in 1st Iteration (a) and focused on the remaining Iterations (b)

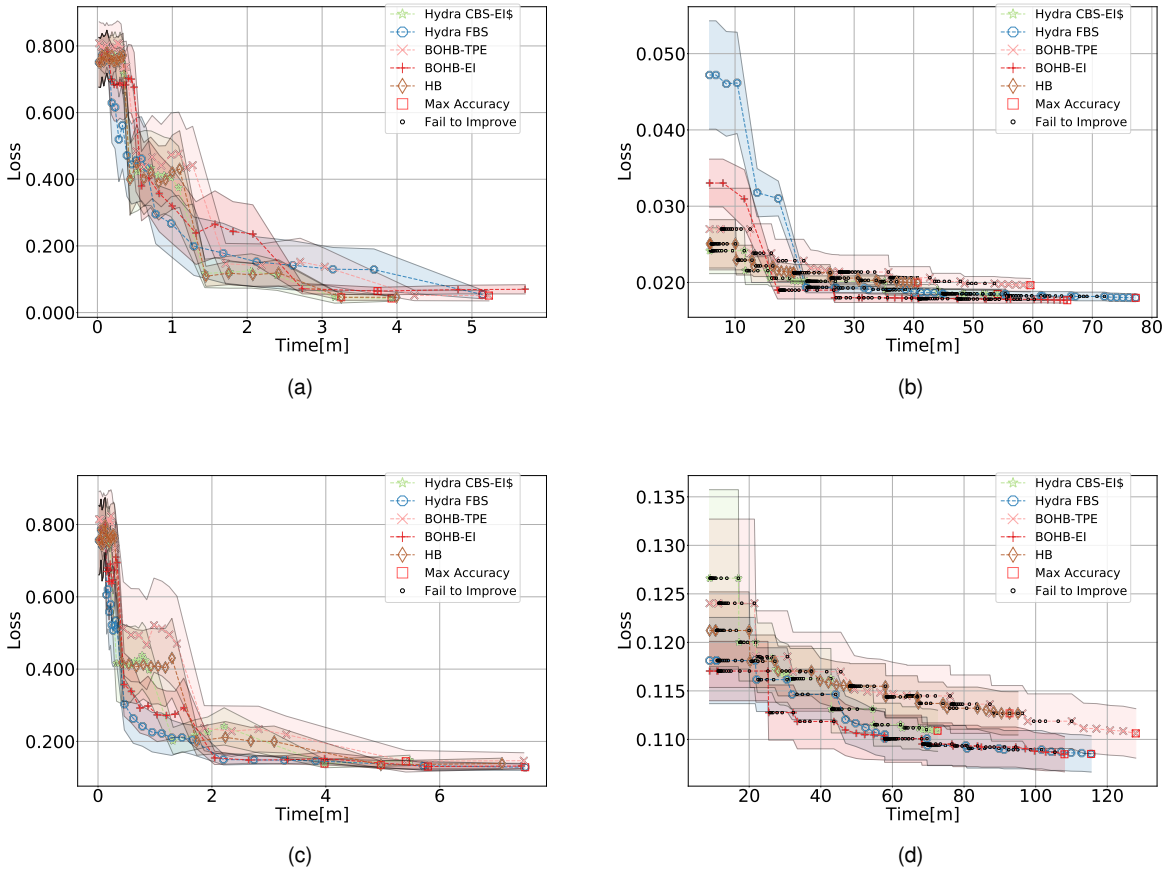
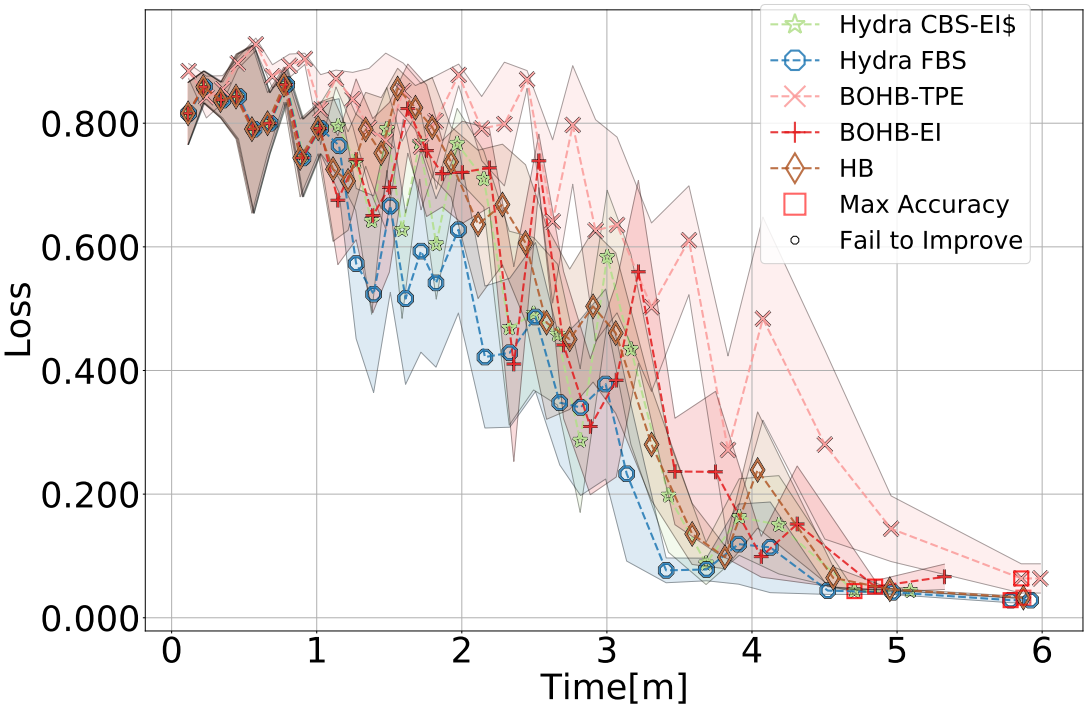
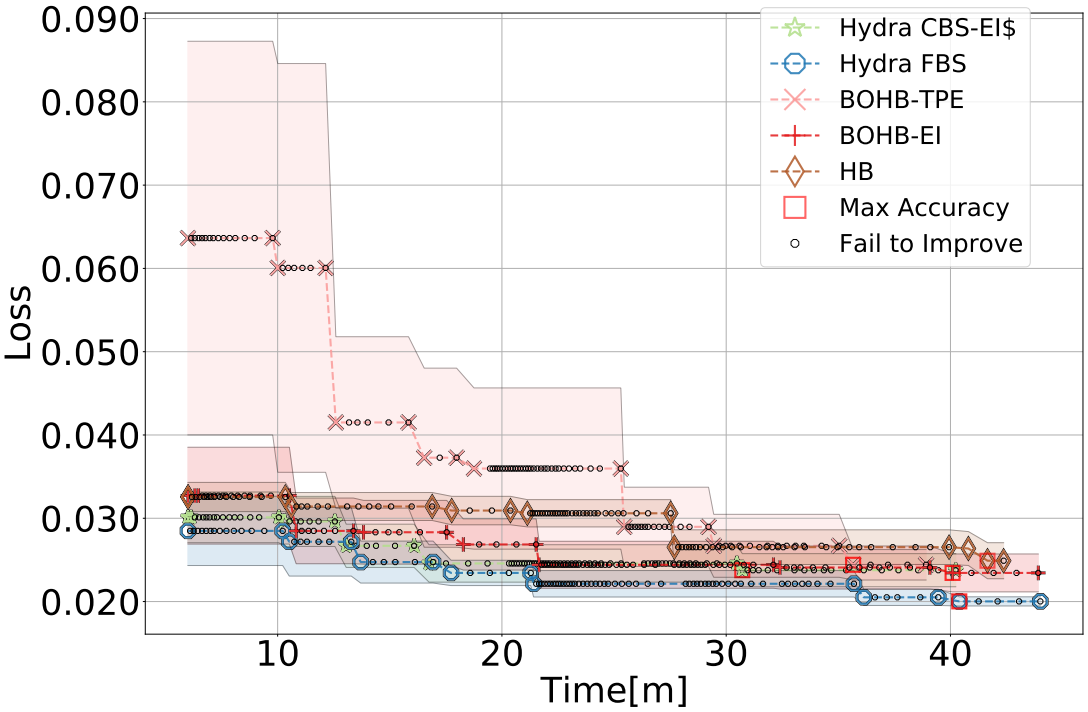


Figure 4.16: Accumulated wall-clock time in minutes in RNN experiment focused in 1st Iteration (a) and focused on the remaining Iterations (b), and in Multilayer again focused on the 1st Iteration (c) and remaining Iterations (d)

MNIST By analyzing figure 4.17 (a) we can note that FBS preserves, also in this experiment, its advantage in the earlier stages of the optimization process. We can also note that in this experiment, this advantage extends throughout all the following iterations (b), producing highly accurate predictions. **In UNET** in figure 4.18 (a) as in figure 4.14 (a) we see all closely matched optimizers in the first iteration. However in (b) we see Hydra FBS achieving the best performance out of all.

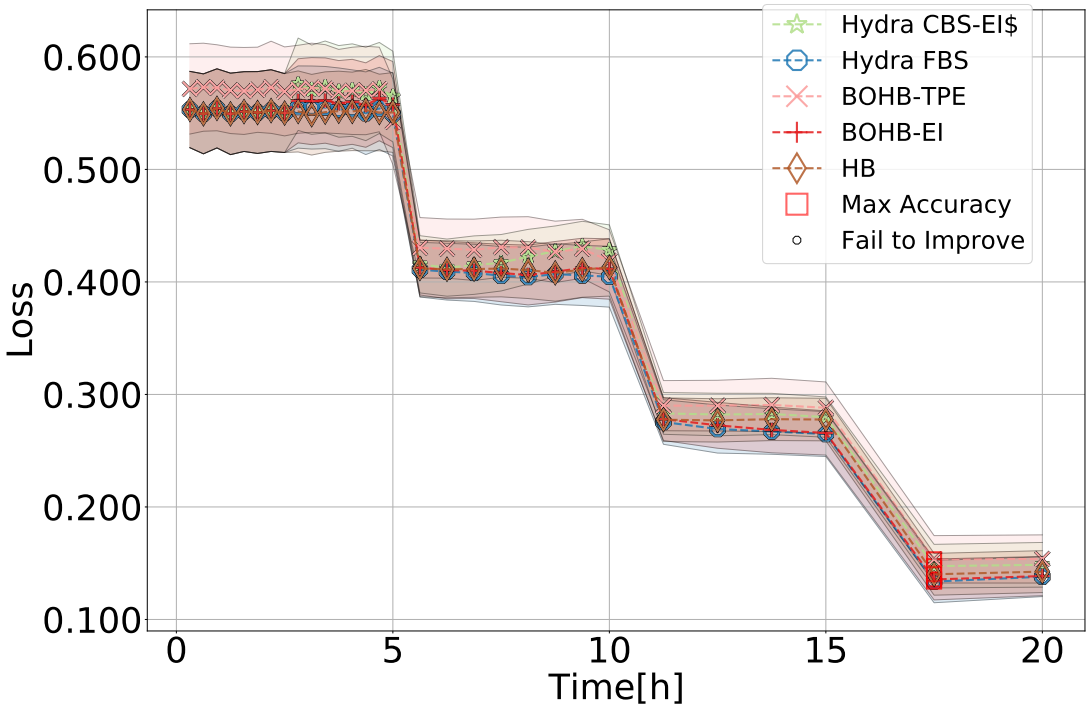


(a)

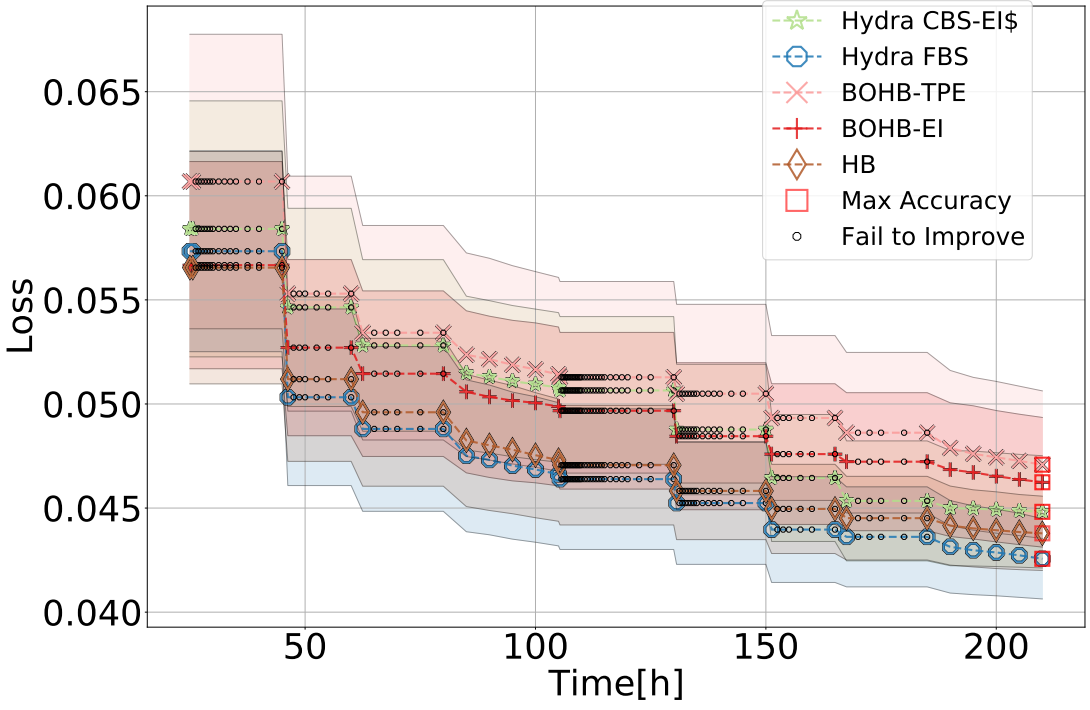


(b)

Figure 4.17: Accumulated Wall-clock time (minutes) and Loss in MNIST, scaled in the 1st iteration, (a) scaled in the 2nd to 10th iterations (b)



(a)



(b)

Figure 4.18: Accumulated Wall-clock time (minutes) and Loss in UNET, scaled in the 1st iteration, (a) scaled in the 2nd to 10th iterations (b)

4.4.4 Summary

In this section we have viewed two of the best-performing variants of Hydra, namely full-budget sampling variant and current-budget sampling with expected improvement per dollar, performing against known state of the art systems such as BOHB with Tree-structured Parzen Estimator and Hyperband. Hydra as table 4.5 shows, has on overhead value that is consistently greater than the other optimizers, however this increase in time becomes insignificant for experiments that spend a great amount of time training the machine learning algorithm. This is evident in experiments such as UNET, CNN, RNN and Multilayer. By contrast, MNIST has major impact, having almost a 22% increase of experiment duration time. Finally, even though this overhead exists and sometimes is significant, the experiments have demonstrated that Hydra achieves constantly lower loss final incumbent values comparing with BOHB-TPE and Hyperband. This is also true in economic-cost focused variants of Hydra, where they can achieve better results with 65% of BOHB-TPE final accumulated cost in economic-cost related experiments (excluding MNIST).

Chapter 5

Conclusions and Future work

Hyperparameter optimization of machine learning is an essential area of artificial intelligence that focuses on enhancing the performance of machine learning models. Unfortunately, though, this process is notorious for being costly and time consuming. Novel state-of-the-art systems regarding this topic have been significantly improving their performance and reducing the associated costs. However, since the majority of largest optimization tasks are performed in the cloud, it is crucial that systems are as fast and efficient as possible, and some, as covered in this report, present shortcomings and miss out on leveraging some techniques that would otherwise improve its performance and efficiency.

This thesis proposes Hydra, a self-tuning system solution that performs optimization of machine learning algorithms improving some drawbacks of previous systems by rapidly converging towards the optimum solution without wasting time on bootstrapping the model, using many low-budget evaluations of configurations while applying transfer-learning to enhance the models' performance, ultimately reducing overall costs.

Hydra achieves consistently higher optimum convergence rates than the extended systems in its full-budget sampling variant in spite of having a slower and more complex model, and lower economic-cost. Comparing with BOHB, Hydra achieves 35% cost reduction while still maintain its speed due to the Hyperband structure, and still outperforms BOHB.

Bibliography

[1] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, n/a(n/a). doi: 10.1002/rob.21918. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21918.

[2] T. Lai and X. Zheng. Machine learning based social media recommendation. In *2015 2nd IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services (ICSDM)*, pages 28–32, July 2015. doi: 10.1109/ICSDM.2015.7298020.

[3] A. KumarGoswami, S. Gakhar, and H. Kaur. Automatic object recognition from satellite images using artificial neural network. *International Journal of Computer Applications*, 95:33–39, 06 2014. doi: 10.5120/16633-6502.

[4] J. A. Cruz and D. S. Wishart. Applications of machine learning in cancer prediction and prognosis. *Cancer Informatics*, 2:117693510600200030, 2006. doi: 10.1177/117693510600200030. URL https://doi.org/10.1177/117693510600200030.

[5] A. Mccallum, K. Nigam, J. Rennie, and K. Seymore. Building domain-specific search engines with machine learning techniques. 04 2009.

[6] P. Mendes, M. Casimiro, P. Romano, and D. Garlan. Trimtuner: Efficient optimization of machine learning jobs in the cloud via sub-sampling, 2020.

[7] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, and W. Zwanepoel. Lynceus: Tuning and provisioning data analytic jobs on a budget, 2019.

[8] D. Didona and P. Romano. Using analytical models to bootstrap machine learning performance predictors. 12 2015. doi: 10.1109/ICPADS.2015.58.

[9] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013. doi: 10.1109/DSN.2013.6575311.

[10] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing*

Systems - Volume 2, NIPS'12, pages 2951–2959, 2012. URL http://dl.acm.org/citation.cfm?id=2999325.2999464.

[11] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, Jan. 2017. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=3122009.3242042.

[12] S. Falkner, A. Klein, and F. Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *ArXiv*, abs/1807.01774, 2018.

[13] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495, 2017. ISBN 978-1-4503-4887-4. doi: 10.1145/3097983.3098043. URL http://doi.acm.org/10.1145/3097983.3098043.

[14] D. Yogatama and G. Mann. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In S. Kaski and J. Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 1077–1085, 22–25 Apr 2014. URL http://proceedings.mlr.press/v33/yogatama14.html.

[15] E. García-Martín, C. F. Rodrigues, G. Riley, and H. Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75 – 88, 2019. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2019.07.007. URL http://www.sciencedirect.com/science/article/pii/S0743731518308773.

[16] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541941. URL http://doi.acm.org/10.1145/2541940.2541941.

[17] C. Delimitrou and C. Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. *SIGPLAN Not.*, 51(4):473–488, Mar. 2016. ISSN 0362-1340. doi: 10.1145/2954679.2872365. URL https://doi.org/10.1145/2954679.2872365.

[18] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 469–482, 2017. ISBN 9781931971379.

[19] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 452–465, 2017. ISBN 9781450350280. doi: 10.1145/3127479.3131614. URL https://doi.org/10.1145/3127479.3131614.

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, 2011. ISBN 978-3-642-25565-6. doi: 10.1007/978-3-642-25566-3_40. URL http://dx.doi.org/10.1007/978-3-642-25566-3_40.

[21] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. 2005. ISBN 026218253X.

[22] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, pages 31–41, 2017. ISBN 978-1-5386-1550-8. doi: 10.1109/SEAMS.2017.11. URL https://doi.org/10.1109/SEAMS.2017.11.

[23] D. Yogatama and G. Mann. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In S. Kaski and J. Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 1077–1085, 22–25 Apr 2014. URL http://proceedings.mlr.press/v33/yogatama14.html.

[24] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor. *Recommender Systems Handbook*. 1st edition, 2010. ISBN 0387858199, 9780387858197.

[25] B. Schafer, B. J, D. Frankowski, Dan, Herlocker, Jon, Shilad, and S. Sen. Collaborative filtering recommender systems. 01 2007.

[26] S. Kotsiantis. Decision trees: A recent overview. *Artificial Intelligence Review*, pages 1–23, 04 2013. doi: 10.1007/s10462-011-9272-4.

[27] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. 2001.

[28] S. Chalup and F. Maire. A study on hill climbing algorithms for neural network training. volume 3, page 2021 Vol. 3, 02 1999. ISBN 0-7803-5536-9. doi: 10.1109/CEC.1999.785522.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 4598:671–80, 1983.

[30] E. Brochu, V. Cora, and N. Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 12 2010.

[31] O. Berger-Tal, J. Nathan, E. Meron, and D. Saltz. The exploration-exploitation dilemma: A multidisciplinary framework. *PLOS ONE*, 9(4):1–8, 04 2014. doi: 10.1371/journal.pone.0095693. URL https://doi.org/10.1371/journal.pone.0095693.

[32] H. J. Kushner. A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. *Journal of Basic Engineering*, 86(1):97–106, 03 1964. ISSN 0021-9223. doi: 10.1115/1.3653121. URL https://doi.org/10.1115/1.3653121.

[33] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, 2014. ISBN 9781931971164.

[34] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. 02 2015.

[35] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2188385.2188395.

[36] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*, volume 54 of *Proceedings of Machine Learning Research*, pages 528–536, Apr. 2017. URL http://proceedings.mlr.press/v54/klein17a.html.

[37] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag. Collaborative hyperparameter tuning. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 199–207, 17–19 Jun 2013. URL http://proceedings.mlr.press/v28/bardenet13.html.

[38] C. Delimitrou and C. Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Transactions on Computer Systems (TOCS)*, 31, 12 2013. doi: 10.1145/2556583.

[39] J. R. Gardner, M. J. Kusner, Z. Xu, K. Q. Weinberger, and J. P. Cunningham. Bayesian optimization with inequality constraints. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–937–II–945, 2014.

[40] L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, Aug. 1996. ISSN 0885-6125. doi: 10.1023/A:1018054314350. URL https://doi.org/10.1023/A:1018054314350.

[41] M. G. Genton. Classes of kernels for machine learning: A statistics perspective. *Journal of Machine Learning Research*, 2:299–312, 2001. URL http://www.jmlr.org/papers/v2/genton01a.html.

[42] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1355. URL https://www.aclweb.org/anthology/P19-1355.

[43] D. Jones. A taxonomy of global optimization methods based on response surfaces. *J. of Global Optimization*, 21:345–383, 12 2001. doi: 10.1023/A:1012771025575.

[44] N. Hansen. *The CMA Evolution Strategy: A Comparing Review*, volume 192, pages 75–102. 06 2007. ISBN 978-3-540-29006-3. doi: 10.1007/3-540-32494-1_4.

[45] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

[46] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. doi: 10.1109/ICEngTechnol.2017.8308186.

[47] A. Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018. URL `http://arxiv.org/abs/1808.03314`.

[48] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. *Deep Big Multilayer Perceptrons for Digit Recognition*, pages 581–598. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_31. URL `https://doi.org/10.1007/978-3-642-35289-8_31`.

[49] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[50] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection, 2017.