TÉCNICO LISBOA

Leveraging Subsampling Techniques to Optimize Machine Learning Jobs in the Cloud

Pedro Gonçalo Bravo Mendes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. João Nuno de Oliveira e Silva
Prof. Paolo Romano

Examination Committee

Chairperson:
Supervisor:
Member of the Committee:

November 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgements

Firstly, I would like to thank my advisors, Professor Paolo Romano, and Professor João Nuno Silva, for their help and support during this thesis. Specially, I am very grateful to Professor Paolo Romano for the continue and constant guidance in the weekly meetings.

Secondly, I thank Maria Casimiro that has a fundamental role in this work, offer me endless support during the thesis, and was always available to clarify my doubts and help me in the most challenging moments. I also thank Professor Luís Rodrigues, who was one of those responsible for choosing this thesis.

I thank my family and in particular, my parents that always support me and gave me the best opportunities throughout my studies and my life. I also thank my sister that, despite all the fights, has an important role in my decisions.

A special thanks to my friends, specially to the group SIM, who was the best support during the last years. In particular, I thank Miguel Pinho, Miguel Malaca, and Rui Cardoso, who were always available to help and to clarify my enormous doubts and were the best partners to work with. Lastly, I thank my colleagues from GSD, who made the many hours of work in room 501 much better.

Lisbon, November 2019
Pedro Gonçalo Mendes

For my Parents,

Resumo

Esta dissertação aborda o problema da otimização do treino de modelos de aprendizagem automática na nuvem. A eficiência destes treinos é afetada pela correta seleção de um grande número de parâmetros de configuração, pertencentes a duas classes principais: os hiperparâmetros, ou seja, os parâmetros do modelo e/ou do algoritmo utilizado no treino; e a escolha correta do tipo e número de recursos na nuvem utilizados para executar o trabalho. Em geral, trata-se de uma otimização com um vasto espaço de configurações, onde soluções sub-ótimas podem levar a uma degradação substancial da precisão do modelo e/ou do aumento dos custos económicos incorridos na aquisição dos recursos na nuvem.

Técnicas do estado de arte abordam este problema empregando abordagens baseadas na Otimização Bayesiana (OB). OB constrói um modelo de caixa negra do desempenho do sistema de forma iterativa. Em cada iteração, o modelo é usado para guiar a seleção da configuração a testar em seguida, usando uma função de aquisição que normalmente equilibra comportamentos exploradores versus exploratórios; a configuração selecionada é então testada e a informação sobre a sua precisão é retroalimentada ao modelo, para melhorar o seu conhecimento e a qualidade das etapas futuras de exploração.

Neste contexto, a presente dissertação investiga o uso de técnicas de subamostragem (ou seja, a redução da quantidade de dados sobre os quais os modelos são treinados) para aumentar a eficiência de duas formas alternativas:

- reduzir o custo do treino ajustando a taxa de subamostragem do conjunto de dados de entrada para compensar, de forma controlada, a precisão dos modelos resultantes e a exigência computacional do processo de treino. Mais detalhadamente, é considerado o problema de minimizar o custo de treino de um modelo de aprendizagem automática, sujeito a restrições de precisão e tempo de execução, enquanto se considera a taxa de subamostragem como uma dimensão adicional do espaço de configuração original. Esse problema de otimização é resolvido através de um otimizador baseado em OB, chamado Nephele. Usando um extenso conjunto de medições obtido pelo treino de três modelos de redes neurais no serviço de computação na nuvem da Amazon EC2 usando um grande número de configurações alternativas, demonstrámos experimentalmente que o sistema Nephele pode alcançar reduções substanciais de custo através de subamostragem (por exemplo, é possivel reduzir 4 vezes o custo, usando uma precisão de 85% e o conjunto de dados do MNIST). Também mostramos que, apesar da inclusão da taxa de subamostragem no espaço de configuração levar a um aumento da dimensionalidade do problema, o custo do processo de otimização do sistema Nephele é comparável, e muitas vezes até inferior, ao custo de métodos equivalentes baseados em OB que não incluem subamostragem no seu espaço de configuração;
- reduzir o custo das técnicas de otimização baseadas em OB através da diminuição do custo nas configurações testadas, devido ao uso de subamostragem. Especificamente, é considerado um problema da otimização na precisão de modelos treinados com o conjunto de dados completo e sujeitos a restrições no custo máximo de treino, e usando apenas conjuntos de dados sub-

amostragem para a avaliação. Este problema é resolvido pelo sistema Fabulinus, um sistema que usa uma nova função de aquisição que seleciona a configuração e o tamanho do conjunto de dados para testar, baseado em dois fatores: i) maximizar a informação sobre a configuração optima no conjunto de dados completo normalizada pelo custo de teste da configuração, ii) maximizar a probabilidade de que a configuração recomendada respeite a restrição de custo. Mostramos que o sitema Fabulinus pode reduzir 6.6 vezes custo de otimização quando comparado com as técnicas clássicas de OB que não usam subamostragem, enquanto aplicam as restrições de custo especificadas, ao contrário das recentes técnicas do estado de arte que usam subamostragem.

Abstract

This dissertation addresses the problem of optimizing the training of machine learning models in the cloud. The efficiency of these jobs be affected by the correct tuning of a large number of configuration parameters, belonging to two main classes: on the one hand, the, so called, hyperparameters, i.e., the parameters of the model and/or of the algorithm used to train the model; on the other hand, the choice of the right type and number of cloud resources used to execute the job. Overall, this is an optimization with a vast configuration space and where suboptimal solutions can lead to substantial degradation of the model's accuracy and/or amplification of the economical costs incurred for acquiring cloud resources.

State-of-the-art techniques address this problem by employing approaches based on Bayesian Optimization (BO). In a nutshell, BO builds a black-box model of the system's performance in an iterative fashion. At each iteration, the model is used to guide the selection of which configuration to test next, via a so called *acquisition function* that typically balances exploitative vs. explorative behaviors; the selected configuration is then tested and information on its quality is fed back to the model, so to improve its knowledge and the quality of future exploration steps.

In this context, this dissertation investigates the use of subsampling techniques (i.e., reducing the amount of data over which models are trained) to enhance efficiency in two alternative ways:

- reducing the training cost by adjusting the subsampling rate of the input dataset to trade off, in a controlled way, the accuracy of the resulting models and the computational demand (and, hence, the cost) of the training process. More in detail, we consider the problem of minimizing the cost of training a machine learning model, subject to constraints on accuracy and execution time, while treating the subsampling rate as an additional dimension of the original configuration space. We solve this optimization problem by means of BO-based optimizer, which we called Nephele. By means of an extensive sets of measurements gathered by training 3 neural network models on the Amazon EC2 cloud in a large number of alternative configurations, we experimentally show that Nephele can achieve substantial cost reductions by trading off accuracy via subsampling (e.g., up to 4 times lower costs, if one accepts accuracy level of 85% using the MNIST dataset). We also show that, despite the inclusion of the subsampling rate in the configuration space leads to an increase of the problem's dimensionality, the cost of Nephele' optimization process is comparable to, and often even lower than, that of equivalent BO-based methods that do not include subsampling in their configuration space.

- reducing the cost of BO-based optimization techniques by decreasing, thanks to subsampling, the cost of testing configurations. Specifically, we consider the problem of optimizing the accuracy of models trained with the full dataset and subject to constraints on the maximum training cost, while relying only on the evaluation of subsampled datasets. We tackle this problem by introducing Fabulinus, a system that uses a novel acquisition function that selects the configuration and dataset size to test by keeping into account two factors: i) maximizing information on the loss-minimizing configuration on the full dataset per unit cost spent testing configurations, and ii) maximizing the likelihood that the recommended configuration will meet the cost constraint. We show that Fabulinus can reduce the optimization cost by a factor up to 6.6 times when compared to classic BO-techniques that do not use sub-sampling, while effectively enforcing the specified cost constraints, unlike recent state-of-the-art techniques that use sub-sampling.

Palavras Chave
Keywords

Palavras Chave

Computação na Nuvem; Optimização de Aplicações; Aprendizagem Automática; Subamostragem; Custo de Exploração; Custo de Produção.

Keywords

Cloud Computing; Optimization of Applications; Machine Learning; Subsampling; Exploration Cost; Cost in Production.

Contents

List of Figures

List of Tables

Acronyms

AWS Amazon Web Services

BO Bayesian Optimization

CEA Constrained Expected Accuracy

CNO Cost Normalized with respect to the Optimum

CDF Cumulative Distribution Function

CF Collaborative Filtering

CNN Convolutional Neural Network

DP Dynamic Programming

EC2 Elastic Compute Cloud

EI Expected Improvement

ES Entropy Search

EI$_C$ constrained Expected Improvement

ES$_C$ constrained Entropy Search

GCE Google Compute Engine

GPU Graphics Processing Unit

HB Hyperband

HPO Hyperparameter Optimization

LCB Lower Confidence Bound

LHS Latin Hyper-Cube Sampling

LSTM Long Short-Term Memory

ML Machine Learning

MTBO Multi-Task Bayesian Optimization

MCMC Markov Chain Monte Carlo

NN Neural Network

PES Predictive Entropy Search

PDF Probability Distribution Function

PI Probability of Improvement

QoS Quality of Service

RNN Recurrent Neural Network

SMBO	Sequential Model-Based Optimization
UCB	Upper Confidence Bound
vCPU	virtual CPU
VM	Virtual Machine
GP	Gaussian Process

1 Introduction

Machine Learning (ML) is an area of study that focuses on the development of algorithms, statistical models, and different techniques in order to create self-learning systems. Machine learning systems have the ability to learn without being explicitly programmed for it. These systems try to use the available data in order to make predictions and decisions based on pattern recognition and inferences from the data. Machine learning predictions rely on the available data used to train. Therefore, the quantity and the quality of this data is very important and has a direct impact on the performance of machine learning algorithms. Also, the computation of such algorithms demands a huge number of intensive calculations and, thus, it is required specific hardware like Graphics Processing Unit (GPU) and/or clusters of multi-core machines.

Machine learning has become increasingly popular due to two main reasons: on the one hand, advances in sensing devices and cyber-physical systems have led to generating a sheer volume of data in a broad number of domains, such as computer vision, bio-informatics, and earth observation; on the other hand, the possibility to process these large volumes of data provided by the cloud computing paradigm, which allows for acquiring an apparently unbounded amount of computational and storage resources, in a convenient, pay-only-for-what-you-only-use fashion.

In recent years, several research works have addressed the problem of selecting the hyperparameters of machine learning jobs [26, 48, 54, 49, 87] (e.g., the batch size or the synchronization method used in the training process), while others have tried to solve the problem of allocating the right type and amount of resources in the cloud for a given application [2, 86, 38, 23, 22, 35]. However, recent works have shown that these two sets of parameters are not independent [19, 18]. They need to be jointly optimized in order to select the best configuration for deploying a job in the cloud.

The resulting configuration space, given by the Cartesian product of the cloud-related and model-related parameters, can easily span hundreds or thousands of different configurations. Also, suboptimal configurations can lead both to poor accuracy and to wasting substantial economic resources in the cloud provisioning process.

Given the complexity of building white-box models capable of predicting the impact of so many variables on the accuracy and execution time of the model's training phase, state-of-the-art systems tend to rely on black-box modeling techniques and Bayesian Optimization (BO). With these approaches, a job is deployed in different configurations and the information on the observed configuration's quality (e.g., execution time and/or accuracy) is fed to a black-box learner (e.g., a Gaussian Process or another regressor) to build a model capable of predicting the quality of untested configurations. The extrapolation power of the black-box model is then used to guide the search process, via a so-called acquisition function that normally strives to strike a balance in the exploration vs. exploitation dilemma [16].

1.1 Objectives

In this context, this dissertation investigates the use of subsampling, i.e., reducing the amount of data over which models are trained, as a means to enhance the efficiency of model training in a twofold way:

1. Reducing the training cost by adjusting the subsampling rate of the input dataset to trade-off, in a controlled way, the accuracy of the resulting models and the computational demand (and, hence, the cost) of the training process. More in detail, we consider the problem of minimizing the cost of training a machine learning model, subject to constraints on accuracy and execution time, while treating the subsampling rate as an additional dimension of the original configuration space.

2. Reducing the cost of BO-based optimization techniques by decreasing the cost of testing configurations thanks to the use of subsampling. Specifically, we consider the problem of optimizing the accuracy of models trained using the full dataset and subject to constraints on the maximum training cost, while testing the model using only subsampled datasets.

1.2 Contributions

This dissertation makes two main contributions, Nephele and Fabulinus, two systems that leverage subsampling techniques to optimize the training of machine learning models in the cloud in different ways:

1. Nephele is a BO-based optimizer that pursues the first objective defined in Section 1.1, i.e., it aims to minimize the training cost subject to accuracy and time constraints by leveraging the possibility of using subsampling to trade-off training cost for accuracy. Nephele is a self-tuning system to optimize the cloud resources and specific parameters of ML applications in order to solve this optimization problem. It leverages the use of subsampled datasets to train a given ML job in order to reduce the training cost while ensuring the Quality of Service (QoS). Nephele treats the subsampling rate as an additional dimension of the original configuration space. It leverages BO to model the accuarcy and the cost functions and exploits those to select the next configuration to evaluate, via an acquisition function.

The starting point to address this optimization problem is Lynceus [19]. To the best of our knowledge, Lynceus is the first system to propose the joint optimization of cloud resources and hyperparameters of ML jobs by leveraging advanced optimization techniques (designated lookahead [53]) and incorporating a budget for the exploration phase. The optimization problem defined in Lynceus matches very closely with the first objective define in Section 1.1. It aims to find the optimal configuration that minimizes the training cost subject to user-defined QoS constraints. Lynceus considers the same set of cloud and model parameters, but it does not exploit subsampling.

By means of an extensive sets of measurements gathered by training 3 neural network (NNs) models (Convolutional Neural Network (CNN), Multilayer Neural Network and Recurrent Neural Network (RNN)) on the Amazon EC2 cloud in a large number of alternative configurations, we experimentally show that Nephele can achieve substantial cost reductions by trading off accuracy via subsampling. Nephele can achieve cost reductions of 75% for training a CNN and 52% for training a RNN comparing with Lynceus if one accepts an accuracy level of 85% using the MNIST dataset. However, training the Multilayer NN was not possible to use subsampled datasets to train in order to reduce the training cost and ensure the QoS constraints, and, thus, the price of the final selected configuration is the same comparing with Lynceus. We also show that despite the inclusion of the subsampling rate in the configuration space leads to an increase of the problem's dimensionality, the cost of Nephele' optimization process is comparable to, and often even lower than, that of equivalent BO-based methods that do not include subsampling in their configuration space.

2. Fabulinus is an optimizer that pursues the second objective defined in Section 1.1, i.e., optimizing the accuracy of models trained using the full dataset and subject to constraints on the maximum training cost, while reducing the optimization cost by testing the model using only subsampled datasets. Fabulinus optimize the selection of cloud resources and specific parameters of a ML job in order to maximize the performance to train the full dataset. However, it only evaluates subsampled datasets in order to reduce the optimization cost and time. Through the use of transfer learning techniques, Fabulinus uses the knowledge gained from evaluating small datasets, to reduce the uncertainty about the location and magnitude of the optimum on the full dataset. It leverages BO to evaluate the search space and build accuracy and cost models.

More precisely, Fabulinus extends a recent system, called Fabolas [48], that proposed the use of subsampling to identify the hyperparameters configuration that maximizes accuracy on the full dataset using evaluations based only on subsampled datasets. However, Fabolas considered a simpler variant of the optimization problem targeted by Fabulinus, i.e., unlike Fabulinus, Fabolas does not support the definition of additional constraints on the recommended configuration. Fabolas targets only hyperparameters, while Fabulinus optimizes in a joint way both hyperparameters and cloud parameters. Fabolas extends an acquisition function, called entropy search, that focus on evaluating configurations and small datasets that can offer more information about the location of the optimum on the full dataset per unit cost. The acquisition function proposed in Fabulinus extends the one proposed in Fabolas in order to allow the definition of constraints. It selects the configuration and dataset size to test by keeping into account two factors: i) maximizing information on the loss-minimizing configuration on the full dataset per unit cost spent testing configurations, and ii) maximizing the likelihood that the recommended configuration will meet the cost constraint.

We show that Fabulinus can reduce the optimization cost in 58.34%, 80%, and 83.93% when compared to classic BO-techniques that do not use subsampling to train a CNN, a RNN and a Multilayer NN, respectively, while effectively enforcing the specified cost constraints, unlike recent state-of-the-art techniques that use subsampling.

An additional contribution of this dissertation is represented by the datasets that we gathered and used to evaluate Nephele and Fabulinus. These datasets were obtained by training different neural networks (Convolutional Neural Network (CNN), Multilayer Neural Network, and Recurrent Neural Network (RNN)) in the cloud. The datasets contain all the information, e.g., the accuracy, training time, and cost, among other metrics, about the deployed jobs. The NNs were implemented using the Tensorflow framework [1] developed by Google and deployed in the Amazon Elastic Compute Cloud (EC2). The dataset used to train the NNs was the MNIST database. Each configuration was composed of cloud resources (number and flavor of Virtual Machines (VMs)), specific application parameters (hyperparameters (batch size and learning rate) and synchronism of training), and the dataset size. Thus, each dataset contains 1440 configurations. In order to decrease uncertainty arising from random processes during training (e.g., the selection of images to train in each iteration), each configuration was trained three times, and it was computed the average and standard deviation of all the measurements. The datasets can represent a valuable asset for the community working on the problem of optimizing ML platforms in the cloud.

Nephele and Fabulinus are evaluated using these datasets. Nephele was compered with two state-of-the-art systems, Lynceus and CherryPick [2], because both systems target the problem of optimizing cloud applications in order to minimize the deployment cost. Fabulinus was compared with Fabolas [48] that uses transfer learning techniques to infer the optimal configuration on the full dataset while using subsampling in the optimization process. The results of the proposed systems show that it is possible to use subsampling in order to decrease the cost of the optimization process and the cost in production, ensuring the QoS.

1.3 Structure of the Document

This thesis is structured in seven Chapters. Chapter 2 describes the cloud, modeling and optimization techniques and analyses related work on systems for optimizing ML hyperparameters and/or cloud resources. Chapters 3 and 4 describe Nephele and Fabulinus, respectively. Each chapter details the objectives, architecture, and algorithm implemented. The datasets gathered for this work are described in Chapter 5 and the evaluation of Nephele and Fabulinus is presented in Chapters 6 and 7. Lastly, in Chapter 8, the conclusions of this work are presented as well as possible future work in order to improve the proposed systems.

2 Related Work

This chapter starts by describing, in Section 2.1, the cloud and the different services and resources offered by modern cloud providers, providing a detailed description about the costs and the pricing schemes charged for these services by the three main cloud providers: Amazon, Google and Microsoft. Based on this analysis, it is possible to understand the importance of state-of-the-art systems that aim to tune cloud resources and application parameters in order to reduce the execution cost and/or maximize the performance. Sections 2.2 and 2.3 provide background on modeling and optimization techniques, respectively. These techniques are commonly used by state-of-the-art optimizers for complex software platforms (such as ML platforms and graph processing systems), which are reviewed in Section 2.4.

2.1 Cloud

Cloud computing [59] provides a ubiquitous, convenient and easy-to-use, on-demand network access to a service that contains a shared pool of configurable computing resources, e.g., networks, servers, data storage, applications and services that can be quickly instantiated with a small direct management effort by the user or without service provider interaction.

Cloud providers offer a variety of machines which can be divided into six categories: general purpose, compute optimized, memory optimized, storage optimized, high performance/accelerated computing and GPU optimized machines. For each category, each provider has a different collection of families of machines that are designed and optimized for specific requirements. For example, each machine's family in the same category can have different processors. Each family also offers different specifications such as the size (number of virtual CPUs (vCPUs)), the available memory and storage and network bandwidth, among others. The number of virtual cores available in a VM varies between 1 to 128 vCPUs in Amazon Web Services (AWS), 208 vCPUs in Microsoft and 416 vCPUs in Google Compute Engine (GCE) and normally increasing the size translates into an increase in the available memory.

Google also offers a service [33] that allows users to customize their machines, choosing the processor, number of vCPUs, amount of memory and disk space, depending on certain rules that have to be followed.

Hence, there is a large range of available possibilities offered by each provider, as shown in Table 2.1, which increases the difficulty of choosing the correct machine to get a good performance at reasonable price. Also, taking into account the Google service that allows users to customize machines, there are millions of different possible configurations. Although Amazon and Microsoft do not offer this service, they have a larger collection of machines.

Cloud providers differentiate the acquired resources in two different types, designated on-demand and reserved resources, according to the period those are reserved. On-demand resources do not have any fidelization period and users pay for all the resources used during the period those are allocated.

	Amazon	Google	Microsoft
Categories	5	5	6
Instance Family	3-7	1-2	1-9
Size (vCPUs)	1-128	1-416	1-208
Total of VMs	184	59	160

Table 2.1: Amount of virtual machines of each cloud provider

Reserved resources are rent for a fixed and mandatory period at lower price rate. The main difference is found in the prices applicable to each service. Reserved resources may have big discounts comparing to on-demand prices, however, these imply a fidelization period of one or three years. Normally, billing can use a price rate of per-hour or per-second. When using an on-demand resource there is always a one-minute minimum charge per-instance, even when a resource is used less than a minute.

Cloud providers also offer a service called Spot-Instances in AWS, Low-Priority VMs in Microsoft Azure and Preemptible VMs in Google Cloud that permits the usage of unused resources that are spare compute capacity in the cloud. This service permits the user to save up to 80-90% [3, 32, 60] on cloud compute costs because it is the user that specifies how much he is willing to pay for the requested instance. However, an instance is acquired only when the bid is higher than the current instance's market price specified by the provider for that instance. This means that there are not fixed prices for an instance and these vary depending on the available offer. When the market price becomes higher than the user's bid the instance stop. These instances can be automatically revoked when there are not enough available Spot-Instances to meet the request or when the providers need those resources. Also, an instance can be stopped, if the request includes a constraint (e.g. time, launch group or an availability zone (place where the machines are) group constraint) that is not met.

The operating system and the availability zone also influence the price. The price list, which may depend on all the previous conditions, can have a huge variety of different prices. For example, using AWS, the cheapest instance has a price of 0.0047$ per-hour, the most expensive instance costs 92,576$ per-hour.

To sum up, there is a myriad of instances a user can choose from. When running an application in cloud and in order to decrease the cost of using such services, it is extremely important to choose the correct amount and type of resources to acquire from the cloud provider. Each provider offers a huge amount of different services at different prices and an incorrect choice of these parameters can lead to unsatisfactory performance (in case of under-provisioning) or to incur unnecessary economical costs (in the case of over-provisioning) So, it is very important to correctly select these resources to get a good enough performance at the lowest possible price. Furthermore, the problem is exacerbated due to the fact that a job's performance for each configuration is unknown *a priori*. This knowledge can only be gain after actually running the job in a configuration, which entails a cost (monetary and temporal) that cannot be ignored. If a bad configuration is chosen, this will be only known after allocating the resources and running the workload. Therefore, it is very important to correctly choose the resources to allocate to run a job in order to get good enough performance at lower prices. The problem of determining an optimum configuration can be very challenging, which motivates the need for systems that can automatically select the previous parameters to minimize the final cost the user pays, while ensuring that the specified constraints are complied with.

2.2 Modeling Techniques

State-of-the-art optimizers for machine learning platforms tend to rely on black-box methods to build models that can be used to predict metrics such as model's accuracy, execution time and cost. This section describes two of the most commonly employed black-box modelling techniques, namely Gaussian Processes (Section 2.2.1) and Decision Trees (Section 2.2.2).

2.2.1 Gaussian Processes

Gaussian Processes (GPs) are stochastic processes [73, 85, 72] defined over a collection of random variables that follow a multivariate Gaussian distribution. GPs are completely specified by the mean function $m(x)$ and the covariance or kernel function K. In optimization problems, GPs are typically used to model an unknown objective function $f(x)$, which has to be either maximized or minimized.

$$f(x) \sim \mathcal{GP}(m(x), K). \tag{2.1}$$

GP is used to fit the information gathered through the evaluation of points of the search space. Each sampled point is modeled with a Gaussian distribution. GP fits the set of these sampled points that follow a normal distribution, which creates a prior belief (model) of the objective function f.

The predictions produced by a GP for any point of the search space follow a Gaussian distribution. The fitted model has an uncertainty, which is reduced as the points are evaluated. The uncertainty about the search space given by the model can be exploited to balance explorative vs exploitative behaviors during the optimization process, as it will be discussed in Section 2.3.

GPs are often used to represent the prior distribution due to flexibility, analytic tractability and marginalization properties of Normal distributions. However, GPs can be challenging and tricky to implement. When implementing a GP model, firstly, a covariance function needs to be chosen and secondly, the hyperparameters of GP need to be tuned, which can be hard to select but has extreme importance in the model's performance.

2.2.1.1 Covariance Functions

The covariance function or kernel has high importance for the GPs because it determines the smoothness properties of points predicted by the model. There are a variety of different kernel functions in the literature that encode the assumptions about the modeled function in different ways.

The squared exponential covariance function is given by

$$k(x_i, x_j) = exp\left(-\frac{||x_i - x_j||^2}{2l^2}\right), \tag{2.2}$$

where the hyperparameter l is the characteristic length-scale, which controls the width of the kernel. This kernel is infinitely differentiable and is very smooth, which can cause unrealistic model's predictions [79].

Another very common class of covariance functions used is the Matérn Class [58]. The kernel functions of this class are given by

$$k(x_i, x_j) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}}{l} ||x_i - x_j|| \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}}{l} ||x_i - x_j|| \right), \tag{2.3}$$

where K_ν is a modified Bessel function and ν and l are positive parameters. ν is a tunable smoothness parameter to control the flexibility of the model and for the most common Matérn functions the parameter ν has a value of $3/2$ and $5/2$. For example, using $\nu = 5/2$, the Matérn kernel function is simplified and is given by

$$k_{\nu=5/2}(r) = \left(1 + \frac{\sqrt{5}r}{l} + \frac{5r^2}{3l^2} \right) exp \left(-\frac{\sqrt{5}r}{l} \right), \tag{2.4}$$

where r is the squared euclidean distance between x_i and x_j, i.e., $r = ||x_i - x_j||^2$.

The aforementioned kernel functions are the most commonly used for GPs due to their properties and simplicity, however, there are several more covariance functions in the literature, like the Polynomial, γ-exponential and Rational Quadratic Covariance Functions [73].

Normally, covariance functions can be multiplied by the signal variance σ_f^2, which is a scaling factor that determines the variation of the values from the mean. Also, when considering noisy observations, the kernel function has additive noise variance σ_n^2.

$$k_y(x_i, x_j) = \sigma_f^2 k(x_i, x_j) + \sigma_n^2 \delta_{ij}, \tag{2.5}$$

where δ_{ij} is a Kronecker delta given by

$$\sigma_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

2.2.1.2 Tuning of Hyperparameter of the Gaussian Processes

The kernel hyperparameters (e.g. the length-characteristic scale l and signal variance σ_f^2) are very important for the model's performance. The length-characteristic scale controls the smoothness of the function and the signal variance determines the variation of points from their mean. However, tuning these hyperparameters is not easy and normally the best hyperparameter settings change for different datasets.

One approach to hyperparameter tuning is to maximize the logarithmic marginal likelihood $p(y|X)$ [73], which has a closed-form expression given by

$$\log p(y|X) = -\frac{1}{2} y^T \left(K + \sigma_n^2 I \right)^{-1} y - \frac{1}{2} \log |K + \sigma_n^2 I| - \frac{n}{2} \log 2\pi. \tag{2.7}$$

A different variant tries to estimate the hyperparameters from a prior by calculating the maximum a posteriori estimate [27], i.e., it selects the hyperparameters that maximize the posterior.

Another approach tries to construct the posterior distribution on f by marginalizing over all possible hyperparameters' values. Normally, to solve this problem, iterative methods for sampling must be used.

One common method is Markov Chain Monte Carlo (MCMC) [31], in particular, the Metropolis-Hastings or Scile sampling algorithm [82, 57, 64], where the prior is used to sample. Another variant uses the conditional likelihood instead of the prior [64]. After the iterative sampling to construct the posterior distribution through MCMC, the hyperparameters are chosen by maximizing the reconstructed posterior.

2.2.2 Decision Trees

Decision Tree [12, 68] is a black-box modeling technique that uses a specific type of graphs, called trees, as a model for predictions (regressions and classifications). Each node specifies a test of some attribute, and each branch descending from that node corresponds to distinct possible values for the attribute [62]. A decision tree is constructed using a direct graph and it is composed of 3 different types of nodes, designated decision, chance and terminal nodes [45] that correspond to different stages of a sequential decision problem. The decision node works as a flowchart-like structure where decisions are made to select an action according to fixed characteristics that will determine the path. The chance node is a node associated with an undefined event, which has some associated probability (e.g., probability of success or failure). The terminal node represents the end of the decision process, where a state prediction is done. A prediction using a decision tree creates a path that starts on the root node, which represents the entire population or space. Using information from observations/evaluations, decisions have to be done in the branches that lead to the final state prediction. This way, a model for classification or regression problems is created.

Most algorithms for building decision trees rely on a recursive on a recursive divide-and-conquer algorithm [14]. It starts with a set of features and the correspondent values and using some metrics (e.g., the Gini Index or the cross entropy), it determines the most representative feature which will be the root, through a measure of impurity. Then, the root is split into two sub-nodes (binary splitting). For each level of the tree, it is calculated the first step again to determine the new most representative feature and a new split is done, recursively. In the end, all the features must be used to create the decision tree. The task of growing a tree can use a different criterion for making the binary splits. The most common measures of impurity [61, 71] are the Misclassification error (Classification Error Rate), Gini Index and Cross-Entropy (information gain).

Misclassification error calculates the prediction error (number of wrong predictions) and tries to minimize the misclassifications. It is calculated by

$$i(m) = 1 - \max_k P(k|m), \tag{2.8}$$

where m corresponds to a leaf (node), $i(m)$ is the impurity of node m and k is a class or value predicted. $P(k|m)$ is the probability of class k predicted in node m. It corresponds to the fraction of training observations in a region that do not belong to the most common class. This measure is sometimes used in classification problems.

Gini Index [12] is an impurity-based criterion to measure the (im)purity of a leaf. It is given by

$$i(m) = -\sum_{k=1}^{K} P(k|m)\ (1 - P(k|m)). \tag{2.9}$$

When a node presents a high value for Gini Index, it means that it has high impurity. Instead, when a node is pure, i.e., the node represents a majority of elements of the same type, the Gini Index has a small value.

Information Gain [69, 62] is another popular measure to calculate the homogeneity of a sample. It computes the quantity of information that can be gained from a feature regarding the class or the final model prediction, i.e. the feature's aid for the final prediction of the ensemble. In a particular work [69] was purposed an algorithm called ID3 that uses cross entropy to calculate the information gain. Cross entropy is given by

$$i(m) = -\sum_{k=1}^{K} P(k|m)\ \log_2 P(k|m) \tag{2.10}$$

If the sample is completely homogeneous the entropy is zero and if the sample is equally divided it has an entropy of one. During the splits in the construction of the tree, to achieve the best performance, the algorithm aims at maximizing the entropy.

Also, there is a technique designated Prunning that aims at reducing the size of decision trees without losing the model's performance and at preventing overfitting by removing sub-sections of a tree that do not provide a relevant contribution to the classification of an instance. Another technique to avoid overfitting issues and therefore an inaccurate model is the implementation of ensemble methods that use several decision trees for predictions. The principle behind the usage of this technique is that a group of different learners working together can form a better and more accurate classifier. There are different methods to construct an ensemble decision trees and the most common are bagging (or bootstrap aggregation) and random forest techniques.

Bagging [11] is a method that uses a set of weak learners (i.e, a learner that has a poor performance) using different subsets of data for training them. After training, the method has an ensemble of different models. The output predictions from different trees need to be reconciled through the implementation of an algorithm for voting or calculating the average predicted value and, in this way, the final prediction of the ensemble is determined. However, as the features used to build each tree are the same and, although the training sets are different, the trees can be overfitted, which may lead to similar results as using one learner.

Random forest [13] is a technique that extends the Bagging algorithm. It is a classifier composed by a collection of trees that selects a random subset of data for training, but it also chooses at random a selection of features rather than using all features to grow trees. This technique aims at reducing the correlation between learners. Each tree may observe different features and the splitting process will lead to different learners, which will therefore output different values.

An ensemble can be used to derive estimators on the uncertainty for a given prediction. A common technique is to assume that the predictions produced by the set of Decision Trees in the ensemble follow a normal distribution, whose parameters can be estimated via the mean and variance of the predictions output by the various learners in the ensemble.

To sum up, the two most common modeling techniques used in the state-of-the-art system to solve BO problems are the Gaussian Process and Decision Trees. GPs are normally used because it outputs a Gaussian distribution by nature. However, the selection of the covariance function and the tuning of the hyperparameteres of the GPs is not a trivial task and has a crucial impact on the model's performance. Also, the training of GPs takes normally much longer [40] comparing with Decision Trees that are fast and easy to train. However, there is no information about the uncertainty of the model. This disadvantages of Decision Trees can be mitigate using bagging ensemble of decision trees or random forests.

2.2.3 Transfer Learning

Transfer learning is a technique that aims at improving the learning process in a target task by leveraging (i.e., transferring) knowledge from previous learned related tasks. There are three common measures that can be used to evaluate whether the use of transfer learning techniques might be beneficial. Firstly, it should be compared the initial accuracy achievable in the target task using a classifier trained only with the transferred knowledge and using an ignorant classifier, i.e., a classifier before training. Then, the training time and the final accuracy should be compared using the transferred knowledge or not. The main goal of transfer learning techniques is to increase accuracy and, at the same time, reduce the training time and cost. However, in order to achieve a positive transfer, i.e., to improve the performance, the tasks must be correlated.

There are two types of transfer learning techniques [84, 6]. The direct transfer learning in which the target task uses the knowledge gain in a previous source task and the multi-task learning in which several tasks are learned simultaneously.

Some works [81, 48] implement transfer learning techniques to leverage small datasets to explore hyperparameter settings for a large dataset. In order to decrease the number of explorations, the knowledge gain by evaluating small subsets of data is transferred to find the best settings on the full dataset.

However, a poor correlation between the two tasks, i.e., if the two tasks are not sufficiently related or if the relationship is not well leveraged by the transfer method, can lead to negative transfer and the performance may fail to improve and decrease its value.

2.3 Optimization Techniques

One of the most common techniques used on state-of-the-art systems and also used in this work is Bayesian Optimization (BO), which is analyzed in detail in this section. Section 2.3.1 introdices the BO technique. Next, the most common acquisition functions used in BO are studied (Section 2.3.1.1) and, finally, rollout techniques for BO are presented.

2.3.1 Bayesian Optimization

Bayesian Optimization (BO) [16, 27, 17, 76, 66] is a model-based method for finding a optimum value of an objective function $f(x)$. Normally, BO is very efficient when $f(x)$ is an unknown black-box function, and when it is possible to gather observation through sampling points from f to build a probabilistic model of the unknown objective function. BO is normally used when the objective function is non-convex, non-linear and it is impossible to calculate the derivatives (BO is a derivative-free method). Usually, the evaluation of f is very expensive, which does it preferable and cheaper to determine and sample points where it will be closer to find the optimum. BO relies on the knowledge (and uncertainty) of the model to guide the exploration of the search space towards the optimum.

BO constructs a model for f using the points observed and exploits this model to determine the next point to evaluate. After each observation, the model is updated with the new data. This ability to incorporate the prior knowledge of f to determine the points to sample yields a very efficient method that requires a relatively small number of explorations to converge to the optimum.

Algorithm 1 SMBO

```
1: function SMBO($n$)                                ▷ $n$: number of initial points to sample and construct the model
2:     Evaluate $f$ on $n$ initial points;
3:     $\mathcal{S} \leftarrow \{x_i, f(x_i)\}$;
4:     while Stopping Criteria is false do
5:         $\mathcal{M} \leftarrow$ FitModel($\mathcal{S}$);                 ▷ Fit a model to the available dara
6:         $x_m \leftarrow$ AcquisitionFunction($\mathcal{M}$);             ▷ Chose the next point to sample
7:         Evaluate $f(x_m)$;
8:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{x_m, f(x_m)\}$;         ▷ Add new data to the training set
9:     return (Incumbent $x_{inc}$)
```

BO uses Bayes' Theorem [80] to update the model. This theorem defines the probability of an event given evidence. The posterior probability of a model given evidence (observation) is proportional to the likelihood of the evidence given the model times the prior belief of the model. The prior represents the belief about the potential space of objective function. Bayes' theorem is used to update the probability for a model as more samples are acquired and information becomes available, thus determining the posterior and updating the beliefs about the objective function.

The prior belief has to be chosen over a set of functions that can describe assumptions about the objective function. Also, in order to efficiently select the point to sample and decrease the number of explorations required to converge to the optimum, an acquisition function is used to guide the search.

Sequential Model-Based Optimization (SMBO) [39] is an algorithm that uses BO methods to fit the models of a black-box function and then leverages these models to choose which configurations should be explored. SMBO is a sequential process that iterates between fitting a model using the existing data, selecting a list of promising points based on this model's prediction, evaluate the target function f on selected points to gather new data and updating the models with the observed experimental values. It extrapolates information about unseen points using the observed ones. SMBO algorithm behaves as described in Algorithm 1. It starts with an initial set of points x_n and evaluates these obtaining $f(x_n)$. A training set $\mathcal{S}$ is created. During each iteration, the prior is updated through Bayes' Theorem to obtain the posterior distribution conditioned on the current training set that contains all the evaluated points until the current iteration. Using an acquisition function, the next point to evaluate x_m is selected. Then, $f(x_m)$ is obtained by evaluating f on x_m. The pair $(x_m, f(x_m))$ is added to the training set $\mathcal{S}$ and thus the new training set for the next iteration is defined.

Gaussian Processes (cf. Section 2.2.1) are often used to build the model exploited in BO [16, 7] due to their flexibility and tractability providing a way to compute the solution in closed-form [76]. This means that each prediction is described as a Normal Distribution with mean value μ and standard deviation σ, i.e., $f(x) \sim \mathcal{N}(\mu, \sigma)$.

The algorithm is running until some stopping criteria is fulfilled, for example, when an available budget or time is exceeded or until the improvement over the current best point attained by further explorations is predicted to be below a given threshold.

However, this algorithm is highly dependent on the constructed probabilistic model and therefore fitting an inaccurate model using the gathered data (including the initial samples) can lead to bad performance and consequently to the algorithm's divergence.

2.3.1.1 Acquisition functions

In order to decrease the number of iterations required to converge, samples should be gathered efficiently. BO uses an acquisition function to select the next point to be evaluated. However, it should be considered if it is preferable to explore unknown regions with high uncertainty, i.e., regions that have a high standard deviation, or if it is preferable to exploit regions that are predicted to have values closer to the optimum. Normally, the most common acquisition functions (Probability of Improvement [51], Expected Improvement [63], Confidence Bound Criteria [21, 4], Entropy Search [36]) provide a trade-off between exploration and exploitation.

Probability of Improvement (PI) aims to maximize the probability of improvement over the incumbent x^*. The incumbent is the predicted optimal configuration and it is updated in each iteration. The goal is to select a point that has the highest probability of having a value closer to the optimum. Considering that the predictions can be described as a Gaussian distribution with mean $\mu(x)$ and standard deviation $\sigma(x)$, PI is given by

$$PI(x) = P(f(x) \geq f(x^*)) = \Phi\left(\frac{\mu(x) - f(x^*)}{\sigma(x)}\right), \tag{2.11}$$

where $\Phi(\cdot)$ is a normal Cumulative Distribution Function (CDF). PI has the advantage of trying to improve more over the current incumbent. However, PI only cares about exploiting and trying to find better values than the current one. So, PI will select points that are predicted with high probability to offer an improvement even if this improvement is very small, instead of selecting points that can offer a better knowledge about the objective function and the location of the global optimum but have high uncertainty. Since a prediction follows a normal distribution, points with higher uncertainty have a lower probability of improvement.

In order to attenuate this drawback, there is a variation of PI [83, 43, 56] where it is added a trade-off parameter ξ to balance the fairness between exploitation and exploration. So, this acquisition function will select points that have a probability of improvement of at least ξ.

Alternatively, Expected Improvement (EI) [63, 44] balances the local and global search, i.e, the trade-off between exploration and exploitation. EI takes into account not only the probability of improvement of a point over the incumbent but also the absolute value of this improvement. The improvement of a point x over the best current x^* is defined as

$$I(x) = \max\{0, f_{t+1}(x) - f(x^*)\}. \tag{2.12}$$

The value of $f_{t+1}(x)$ can be predicted using the probabilistic model conditioned to all the available information from previous observations.

EI computes the expected value $\mathbb{E}$ of the improvement I and it is given by

$$EI(x) = \mathbb{E}[I(x)] = \int_{-\infty}^{f(x^*)} (f(x^*) - c)\, P_M(c|x) dc, \tag{2.13}$$

where the variable c is defined over the same domain as $f(x)$ (namely, the objective function that we intend to minimize) and $P_M(c|x)$ is the probability for configuration x to have ”performance” c, as predicted by the model. If the prediction for $f(x)$ is described by a normal distribution with predicted mean value $\mu(x)$ and the associated uncertainty, the EI has a closed-form expression given by

$$EI(x) = \begin{cases} (\mu(x) - f(x^*))\, \Phi(Z) + \sigma(x)\phi(x)\ , & \text{if } \sigma(x) > 0 \\ 0\ , & \text{if } \sigma(x) = 0 \end{cases} \tag{2.14}$$

where

$$Z = \frac{\mu(x) - f(x^*)}{\sigma(x)}, \tag{2.15}$$

$\phi(\cdot)$ and $\Phi(\cdot)$ are the Probability Distribution Function (PDF) and CDF of the standard normal distribution, respectively.

In order to extend EI to incorporate constraints on the optimization problem, a variation of this acquisition function, called constrained Expected Improvement (EI_C) [28], was purposed. EI_C considers that the incumbent is the point with the optimum value but which respects all constraints. Also, the predicted infeasible points, i.e., points that do not respect the constraints, have an improvement of 0. The constrained improvement is given by

$$I_C(x) = \delta(x) \max\{0, f_{t-1}(x) - f(x^*)\} = \delta(x) I(x), \tag{2.16}$$

where $\delta(x)$ is a binary factor ($\delta \in \{0, 1\}$) to indicate the feasibility of a point. $\delta(x)$ can be described as a Bernoulli random variable. So, the expected value of $\delta(x)$ is $\mathbb{E}[\delta(x)] = P(\delta(x) = 1) \cdot 1 + P(\delta(x) = 0) \cdot 0 = P(\delta(x) = 1)$, i.e., the probability of a point X meeting the constraints. Also, $\delta(x)$ and $I(x)$ are conditional independent events, given x. The constrained expected improvement is then given by the following expression.

$$EI_C(x) = \mathbb{E}[I_C(x)] = \mathbb{E}[\delta(x)I(x)] = \mathbb{E}[\delta(x)]\,\mathbb{E}[I(x)] = P(\delta(x) = 1) \cdot EI(x). \tag{2.17}$$

Two other acquisition functions, designated Lower Confidence Bound (LCB), which is used to minimize the objective function, and Upper Confidence Bound (UCB), which is used to maximize the objective function [21, 4], select points to sample through a confidence bound criteria.

$$\begin{cases} LCB(x) = \mu(x) - \kappa\sigma(x) \\ UCB(x) = \mu(x) + \kappa\sigma(x) \end{cases} \tag{2.18}$$

where $\kappa \geq 0$ is a tunable parameter to control the balance between exploration and exploitation. These acquisition functions aim to minimize the regret during the optimization process. There is a variant of the UCB called GP-UCB [77] that presents an approach to select the parameter κ. An instantaneous regret function $r(x) = f(x^*) - f(x)$ is defined and the goal is to minimize the cumulative regret. Using the UCB criterion, GP-UCB is defined as

$$GP\text{-}UCB(x) = \mu(x) + \sqrt{\nu\beta_t}\sigma(x), \tag{2.19}$$

where $\nu > 0$ is a hyperparameter and β_t are appropriate constants. The next point to sample is selected maximizing the GP-UCB.

Entropy Search (ES) [36] adopts a strategy of evaluating points that are expected to offer more information about the optimum, i.e., based on the predicted information gain about the optimum, instead of selecting points predicted to be near to the optimum. The sampled points are estimated to most decrease the entropy of the distribution, i.e., decrease the uncertainty over the location of the optimum. The belief about the optimum given the prior on f and the training set $\mathcal{S}$ is given by the probability distribution $p_{min}(x|\mathcal{S}) = p(x \in \arg\min_{x'} f(x') \mid \mathcal{S}))$. The information gain at x is calculated using the relative entropy (Kullback-Leibler divergence) [50]. The relative entropy is a measure of the difference between two probability distributions P and U and is given by

$$\mathcal{D}_{KL}(P;U) = \int p(x) \log \frac{p(x)}{u(x)} dx, \tag{2.20}$$

where p and u denote the PDF of P and U, respectively. For the base distribution U is chosen a uniform distribution [36] and it is compared to the probability distribution p_{min}. ES is given by

$$ES(x) = \mathbb{E}_{p(y|x,\mathcal{S})} \left[\int p_{min}(x') \cdot \log \frac{p_{min}(x')}{u(x')} dx' \right]. \tag{2.21}$$

Due to the complexity of computing p_{min} and the integral of Equation 2.21, it is constructed a first-order approximation to calculate the expected information gain from an evaluation x' [36].

ES has the objective of finding a point x from a set of possible candidates that maximizes the information gain over the distribution of the location of the optimum. There is a variant of this acquisition function designated Predictive Entropy Search (PES) [37], where other optimization and approximation techniques are used to compute Equation 2.21.

The acquisition functions previous mentioned are the most common in the literature, in particular, in the state-of-the-art systems that will be reviewed in Section 2.4. Overall, every acquisition functions has both advantages and drawbacks. The PI presents a pure exploitative behavior, thus, it discards points with high uncertainty. To balance the exploration and exploitation trade-off, the EI can be computed to determine the expected value of the improvement. Also, the computation of these acquisition functions is facilitated if one assumes an underlying Gaussian model, as in the case of GP, as this allows computing in closed form several of the above discussed acquisition functions. EI is easy to be extended to incorporate constraints (EI_C). The LCB adopts a more optimistic behavior and gives a confidence bound to the predicted values. This confidence bound is controlled by a tunable parameter to balance exploration and exploitation, however, this parameter is tuned by the user, which may be difficult to select. The ES is another common acquisition function that aims at evaluating configurations that are expected to offer more information about the optimum. Nevertheless, the computation of this acquisition function can be time-consuming and requires high computational power.

Lastly, all these acquisition functions are greedy/myopic, i.e., they consider only one step ahead in the optimization process, which, as for all greedy heuristics, can lead to explore the search space in suboptimal ways.

2.3.1.2 Baysian Optimization with Lookahead

Several works present in the literature [29, 30, 52, 65] use BO to minimize the overall cost given a fixed budget for the exploration phase. These works extend BO combining the greedy acquisition functions with lookahead techniques to move towards a long-term reward and, thus, to mitigate the shortcomings of those greedy acquisition functions. In order to implement a lookahead heuristic, a policy, that contains the rules to specify the way of updating the model when there is new information available, and a reward function, which quantifies the benefits of evaluating a new set of points, have to be defined. Normally, the reward function is defined as the maximization of the acquisition function.

The work by Lam and Willcox [53] proposed a BO-based approach that supports lookahaed and can account for additional constraints on the recommended configuration. This algorithm formulates constrained BO as a Dynamic Programming (DP) problem [5]. DP is a backward recursive algorithm. To compute the reward, it is required the resolution of several nested maximizations and expectation with unknown closed-form, from the end to the initial state. Therefore, DP is computationally intractable. To mitigate the cost of solving the DP problem, a rollout technique [67, 9, 53, 52] is implemented, which maximizes the long-term reward to select the next point to evaluate. The reward is calculated by simulating optimization scenarios over future steps. Rollout uses a heuristic policy to select designs to be sampled that approximate the optimal reward by selectively eliminating costly designs. Unlike DP, using rollout the information only propagates to the future steps and each step depends only on the previous ones, regardless of the future steps. This means that rollout is a closed-loop approach that uses the information gained in each step of the simulation to simulate the next stages. Two numerical simplifications are introduced to deal with the nested expectations problem. Firstly, to decrease the number of future steps simulated, a rolling horizon, i.e, a limited finite number of future steps, is implemented. Secondly, the expectations are approximated using Gauss-Hermite (G-H) quadrature [55]. G-H quadrature is used to approximate integrals of the shape $\int_{-\infty}^{\infty} e^{-x^2} f(x)\, dx$, obtaining $\int_{-\infty}^{\infty} e^{-x^2} f(x)\, dx \approx \sum_{i=1}^{N} w_i\, f(x_i)$, where N is number of points to sample and w_i is a weight associated with each sampled point.

Not that, increasing the rolling horizon does not necessarily mean the improvement of the algorithm's performance [53]. The horizon's increase might augment the model's uncertainty because the values added to the model are predicted and these predictions can be incorrect. Hence, new predictions may be less and less accurate. The calculated reward for the different values will be more uncertain and less accurate with the increasing of the horizon.

2.4 Optimizing Parameters of Complex Systems

There are several different types of tuning systems, with different objectives, for different applications. In this section, two different categories of state-of-the-art systems are introduced, explained and discussed. Firstly, systems for optimizing hyperparamters of machine learning models are presented (Section 2.4.1). Next, we review systems for optimizing the allocation of cloud resources for different type of applications, whose purpose is to maximize the quality of the final output solution and ensure that the Quality of Service (QoS) constraints are met with the minimum possible cost are reviewed (Section 2.4.2). The analysis presented in this work focuses on the following main differentiating factors: i. optimization technique, ii. optimization problem, iii. exploit subsampling, and iv. consider additional constraints.

2.4.1 Optimizing Machine Learning Hyperparameters

There are different systems for tuning hyperparameters of machine learning training processes in order to obtain better accuracy. These systems allow to optimally tune the application-specific parameters of interest and thus obtain better final performances at lower costs.

Bayesian Optimization (cf. Section 2.3.1) is a common technique used in the reviewed state-of-the-art systems [48, 26] to find the best hyperparameters of ML job. These systems are based on SMBO algorithm [39] where a model is built through the evaluation of different configurations in the search space. Next, this model is exploited to select the next configuration to evaluate and to recommend the best configurations. BO is a model-based technique. Therefore, the quality of the predictions are dependent of the previous knowledge. To tackle the limitations of BO, Hyperband was proposed.

Hyperband. Hyperband [54] is an algorithm for hyperparameter optimization that allocates resources (e.g. the number of iterations, dataset sizes or the number of features) to evaluate a configuration. It is based on the Successive Halving algorithm [42] where a budget B is allocated to a set of different configurations. The resources allocated depend on the budget and B/n resources are allocated on average for configurations, where n is the dataset size. Then, these configurations are evaluated and the configurations' set is reduced to half. The 50% of configurations removed from the set correspond to the configurations with the worst performance. This cycle is repeated until one final configuration remains.

Hyperband aims at maximizing the accuracy without considering any constraints using the user-defined maximum amount of resources R that can be allocated for one configuration. The algorithm receives an input value of R and η that defines the portion of configurations rejected in each iteration s. Parameter η defines the aggressiveness with which the algorithm eliminates configurations: higher values correspond to a more aggressive strategy. More aggressive strategies require fewer iterations for the algorithm to reach its stopping condition. Hyperband determines the possible values of the dataset size n , for a fixed budget B, through the computation of the Equation 2.22

$$n = \frac{B\eta^s}{R(s+1)} \tag{2.22}$$

where the budget B is given by $B = (s_{max} + 1)R$. s_{max} is the maximum number of iterations computed through $s_{max} = \log_\eta(R)$. For each n there is a minimum resource r to allocate for each configuration, given by Equation 2.23. To larger values of n correspond lower resources r.

$$r = R\eta^{-s}, \tag{2.23}$$

Then, n configurations are randomly chosen. This set is evaluated on the respective allocated resources. Configurations with the best performance are selected for the next iteration. This procedure is repeated until one configuration remains on the set. Hyperband does not rely on model predictions as the BO-based systems. Also, it discards bad configurations early without spending a huge budget to evaluate them.

Hyperband does not exploits subsampling. Also, the optimization problem solved does not incorporate constraints. However, Hyperband can converge slowly since it requires the random evaluation of a large number of configurations. This problem is emphasized when using large budgets. In order to tackle this drawbacks, a new system, called BOHB was developed that replaces the random selection of configurations at the beginning of each iteration by a model-based search.

BOHB. BOHB [26] is a system to Hyperparameter Optimization (HPO) that combines Bayesian Optimization (BO) and Hyperband (HB) methods. BOHB implements a hyperband method in order to determine the number of configurations to evaluate given a budget. Hyperband [54] is used to allocate the resources in each iteration and based on the Successice Halving algorithm [42] the best configurations contained in a set of randomly sampled configurations are identified and the under-performed configurations are dropped. However, BOHB implements a model-based selection of the configurations to be evaluated, instead of a randomly selection.

BOHB aims is to maximize the accuracy using the largest budgets. It does not exploit subsampling and does not considers constraints. The given budget for each exploration increases with the number of explorations. BOHB always uses the model to predict the performance of configurations for the largest budget. BOHB builds a model for the objective function and uses BO to choose the next configuration to evaluate based on the previous experiments. Instead of GPs to model the objective function, BOHB uses the Tree Parzen Estimator [8] that computes the defined acquisition function (EI) required by the BO method in a more efficient way.

In order to reduce the cost of testing configurations, a state-of-the-art system, called Fabolas, proposed the use of subsampling technique to evaluate cheaper configurations and transfer learning technique to apply the knowledge acquired on smaller datasets on the full dataset.

Fabolas. Fast Bayesian Optimization on Large data Sets (Fabolas) [48] is a system for tuning machine learning hyperparameters. In order to decrease the time of hyperparameter optimization, it uses subsampling to evaluate the performance of smaller datasets sizes, instead of using the full dataset, which normally is expensive to evaluate. Testing different configurations on small datasets permits the extrapolation to the full dataset.

Fabolas adds one additional dimension to the search space that corresponds to the dataset size s to evaluate and is given by N_{sub}/N, where N_{sub} is the size of the subsampled dataset and N is the full dataset size. s is a value between 0 and 1 ($s \in [0, 1]$) that corresponds to the fraction of the dataset used. For example, if $s = 0.5$, half of the dataset is used, if $s = 1$, the entire dataset is used. This parameter is also an input of the black-box functions to optimize and, in this way, the dataset size of the configuration to evaluate is determined automatically by the optimizer.

The goal of Fabolas is to minimize a loss function f (equivalent to maximizing the accuracy) on the full dataset, i.e., for $s = 1$, but evaluating only subsampled datasets. It enables the correlation of performance across dataset size. It uses Gaussian Processes (GPs) to model loss and cost and the GP kernel chosen is the Matérn $5/2$ Kernel [58] multiplied by a finite-rank covariance function in s.

$$k\left((x, s), (x', s')\right) = k_{5/2}(x, x') \cdot \left(\phi^T(s) \cdot \Sigma_\phi \phi(s')\right), \tag{2.24}$$

where ϕ is a basis function that incorporates the behavior of the function across s. For the loss model, the basis functions is given by $\phi_f = (1, (1-s)^2)^T$ because normally loss increases when there are more data available, i.e., for higher values of s. The complexity of the computational cost/time c usually grows with relative dataset size, i.e., $\mathcal{O}(s^\alpha)$, for a arbitrary α and, to enforce positive predictions, the modeled function is the logarithm of the training time. Therefore, for the training time model, the basis function is instead given by $\phi_c = (1, s)^T$. The hyperparameters of the GPs are selected via Markov Chain Monte Carlo (MCMC). To accelerate the process of searching for the optimum, it is used hyper-priors, i.e., prior distributions of the hyperparameters, to highlight parameter values that are believed to be more relevant.

Fabolas uses Bayesian Optimization (BO) to determine the optimum of the optimization problem given by $\underset{x}{\text{minimize}}\, f(x, s = 1)$, where f is the loss function. As an acquisition function, Fabolas uses the Entropy Search (ES) [36] (cf. Section 2.3.1.1) normalized with respect to cost. It tries to maximize the information gain per unit of cost about the distribution of the optimum for $s = 1$, $p_{mim}^{s=1}(x|\mathcal{D}) = p(x \in arg\min_{x' \in \mathbb{X}} f(x', s = 1)|\mathcal{D})$, where $\mathcal{D}$ represents the training set. The acquisition function selects a configuration (x, s) that maximizes the information per unit cost that is possible to gain about the optimum (on s=1). By evaluating and adding to the training set the configuration (x, s), it quantifies the knowledge learned about the performance on the full dataset. The acquisition function, given by Equation 2.25, computes the relative entropy between $p_{min}^{s=1}(x'|\mathcal{D} \cup \{(x, s, y)\})$ and a uniform distribution $u(x')$, with expectations taken over the evaluation y to be obtained at x. It is normalized with respect to the optimal cost and the overhead cost $C_{overhead}$ in order to select configurations with lower cost. Fabolas considers the computational cost, i.e, the time needed to evaluate a configuration or to compute the acquisition function.

$$a_F(x, s) = \frac{1}{C(x, s) + C_{overhead}} ES(x) \tag{2.25}$$

For the initial design, Latin Hyper-Cube Sampling (LHS) is used to select initial configurations and each configuration is evaluated on different datasets sizes. This way, it is provided more information on scaling behavior, i.e., information about the evolution of the function on the s dimension.

In each iteration, Fabolas predicts the loss for all configuration in the training set at $s = 1$ and the incumbent configuration (i.e., the current predicted optimum) is the one that minimizes the loss. The computational cost and resources needed to compute the acquisition function are very high and so the system needs too much time and resources to solve the optimization problem. The use of GPs also increases the execution time of Fabolas. A key limitation of Fabolas is that it does not support the definition of additional constraints on the recommended configuration. In order to tackle this limitation, we proposed Fabulinus that extends Fabolas' acquisition function and the logic it uses for selecting the incumbent configuration.

2.4.2 Optimizing Cloud Resources

Due to the concerns and restrictions discussed on Section 2.1, it is very important to correctly allocate cloud resources for different applications in order to increase the performance and/or meet their QoS constraints at the lowest possible cost. Several works were proposed in this area over the last few years. The existing literature can be distinguished and classified based on a number of criteria. The analysis of the systems considers the same differentiating aspects as in the previous section.

CherryPick. CherryPick [2] is a system that aims to select the optimal cloud configuration that minimizes the execution cost and ensures that the time performance is below a specified constraint. It solves the following optimization problem.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & C(x) = P(x) \cdot T(x) \\ \text{subject to} \quad & T(x) \leq T_{max}; \end{aligned} \tag{2.26}$$

where $C(x)$ is the cost of running a configuration x in the cloud, $T(x)$ is the running time of that configuration, $P(x)$ is the price per unit of time of the allocated resources (VMs) corresponding to configuration x in the cloud and T_{max} is the time threshold to run a configuration, that is, the maximum time allowed for the job to complete. P(x) is known *a priori*, but T(x) is not and varies with x in ways that are not easy to predict.

CherryPick leverages BO to build a performance model in order to predict the best configurations that solve the problem of Equation 2.26. GPs are used as a prior function using the Matérn $5/2$ kernel and the constrained Expected Improvement (EI_C) is used as acquisition function.

A configuration is composed of the number of VMs, CPU count, CPU speed and RAM per core, disk count and speed, and network capacity. To construct an initial model, CherryPick selects three configurations at random. Then, the model's prediction is used to compute the EI_C for all untested configurations and the configuration selected for the next sampling is the one associated with the largest EI_C value. This procedure is repeated until the predicted expected improvement is below a given threshold and N configurations are evaluated. This way, CherryPick guarantees that a minimum number of configurations were observed and also it prevents against insignificant improvements that might be time and cost consuming.

CherryPick has four components: i. the Search Controller, which is responsible for the cloud configuration selection process, ii. the Cloud Monitor that measures the cloud experiments, iii. the BO Engine that manages the optimization process of selecting configurations to sample and constructing the model, and iv. the Cloud Controller that launches and controls the experiments on the cloud.

CherryPick minimizes the cost satisfying a performance constraint. Nonetheless, it does not have into account a notion of an available budget for searching for the optimum, i.e., the exploration cost of all configurations is not considered. Therefore, it does not exploit subsampling in order to reduce the cost. Also, the search space considers only 66 configurations, which is a small number especially if it is taken into account all the possible configurations aforementioned (Section 2.1). The stop condition requires the evaluation of a minimum of 6 configuration, which represents 9% of the search space. Furthermore, the initial model that uses almost 5% of the search space is trained using random samples that may be expensive and underperformed, creating a poor and high uncertain model. Although CherryPick only explores 9% of the search space, which represents a relatively small value of explored configurations to find the optimum, the percentage of configurations explored to create the initial model is high when compared with other state-of-the-art systems, as Lynceus, that use small values.

Lynceus. Lynceus [19, 18] is a budget-aware and long-sighted self-tuning system of cloud resources. It aims at finding optimal configurations that minimize the execution cost of running data analytics jobs in the cloud while ensuring that the constraints on maximum execution time are complied with and that the cost of the exploration phase does not exceed the available budget B.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & C(x) \\ \text{subject to} \quad & T(x) \leq T_{max} \\ & \sum_{x_k \in \mathcal{S}} C(x_k) \leq B \end{aligned} \tag{2.27}$$

where $C(x)$ and $T(x)$ are the execution cost and time of running configuration x for a given job, respectively, $\mathcal{S}$ is the training set that contains all the observed configurations. The cost $C(x)$ of a configuration x can be computed by multiplying the running time of that configuration $T(x)$ by the cost per unit-time $P(x)$ of using the cloud resources that constitute configuration x.

Lynceus is budget-aware because it incorporates a budget for exploration and, in each iteration, dynamically selects the configurations to explore according to the current budget. However, it does not use subsampling in order to reduce the cost. Configurations incorporate not only cloud resources, as VMs' number and type, but also hyperparameters of the machine learning jobs. Lynceus uses a cost model that is built using the measurements collected when testing configurations and is used to predict the cost for untested configurations. At the beginning of the exploration phase, while there is a large available budget and the cost model has high uncertainty, Lynceus presents a more explorative behavior. After exploring more configurations, the model's uncertainty and the budget decrease and Lynceus selects configurations to sample that will not compromise the budget using the model to maximize shorter term reward.

This system uses BO to solve the optimization problem, building a cost model $\mathcal{M}$, using a bagging ensemble of decision trees (Section 2.2.2), trained with a dataset, noted $\mathcal{S}$, containing the tested configurations. A set of initial points is chosen using LHS and the model is trained over this initial set.

Lynceus uses the lookahead technique to predict a path of configurations to be explored that maximizes the reward and minimizes the exploration cost, instead of selecting configurations based on a myopic approach. The lookahead technique aims to foresee the effect of choosing a configuration to evaluate in the current iteration. For that, it simulates the next iterations using predicted values by the model. The model is updated with the predicted cost of the configuration that maximizes the reward and then, in the next lookahead step, it predicts the new reward and cost of untested configurations based on the model updated with predicted values, instead of real values.

In each iteration, Lynceus creates a feasible set Γ that contains the configurations that are predicted with a probability higher than 99% to have a cost that is lower than the available budget. Then, for each configuration x in this feasible set, it computes the reward by calculating the Constrained Expected Improvement (EI_C) that is used as acquisition function and predicts the improvement, over the current known best, brought by exploring a given configuration. The prediction of the configuration that maximizes the reward is used to update the model. The next lookahead step starts using this speculated information and repeats the procedure, since the creation of a new feasible set. Instead of updating the model only with the mean value predicted by the cost model, Lynceus exploits the fact that its underlying model can estimate the probability distribution for the cost and, at each lookadhead step, Lynceus speculates about all the possible cost values, and their predicted likelihood, for the configuration selected in the previous lookahead step. Therefore, the model is cloned and updated with different possible values c_i, as described next. For each cost c_i, the training set is updated $\mathcal{S} = \mathcal{S} \cup \{(x, c_i)\}$, the model is retrained and the budget is decremented by c_i. Since the. number of possible cost values is theoretically infinite, the G-H quadrature is employed to approximate the computation using a (small number of) different cost values. When the lookahead reaches the maximum horizon h or when the feasible set is empty, a so-called exploration path (or simply path) ends. For each path, Lynceus computes the corresponding reward and cost and selects the path that maximizes the ratio of reward to cost. As shown in Lynceus, lookahead can explore more configurations and find solutions with lower cost. However it does not necessarily work better than greedy methods. In particular, given its stronger reliance on the model (to speculate on future explorations), it is less robust to model's mispredictions. Also, it has also higher computational costs than conventional greedy BO, as the computation of the reward of a path grows exponentially with the length of the path. Lynceus can incorporate constraints on execution time: Furthermore, it can be extended to incorporate additional constrains under the assumption that the constraints are independent. In spite of Lynceus aims at minimizing the cost, it does not use subsampling, thus requires expensive evaluations using the full dataset.

PARIS. Performance-Aware Resource Inference System (PARIS) [86] aims to provide an estimate for cost and performance for different VM types with minimal data collection in order to select the right VM type that meets the target performance and cost constraints. However, unlike the two previous systems, it does not use BO.

The system receives as input a representative task of the workload, the performance metrics and a set of candidate VM types. PARIS models the resource requirements of the workload and the behavior that different VM types have on workloads with identical requirement resources, using Random Forests. The modeling task is divided into two phases designated offline and online phases. In the Offline phase, PARIS runs a broad set of benchmarks using different resources and collects detailed systems performance metrics and statistics for each VM type. This phase has a fixed cost and, in the end, models are created using decision trees and trained with the collected data. When there are new VM types, the benchmark only needs to be run on those. The Online phase calls a Fingerprint-Generator that runs a representative task given by the user on 2 reference VM types and collects performance metrics and resource usage information that tries to capture patterns and helps to understand the resource requirements of the task. Using the data collected in both phases a set of decision trees is trained for each workload, obtaining a random forest. Thus, a performance model is obtained.

Paris considers that the cost is a unknown function of the target performance metric (e.g., execution time or throughput) and of the price per hour of the VM. Thus, by estimating the performance of the job via the model, PARIS can obtain a prediction of the cost of choosing those instances. Once again, the model is highly dependent on the correct choice of reference configurations and the available data in the training set. Unlike Lynceus and CherryPick, PARIS requires offline knowledge of previous workloads. This allows for avoiding the need for sampling many configurations when a new workload has to be optimized. However, offline knowledge may not be available and is time consuming and expensive to build. PARIS does not consider the total exploration cost, only the deployment cost of a configuration. Also, it does not exploit subsampling in order to achieve cost reductions.

Scout. Scout was designed in order to provide solutions to some of the drawbacks of CherryPick and PARIS. Scout aims at maximizing the performance of an application while ensuring a constraint on cost. Scout incorporates the search-based method to accommodate mispredictions as CherryPick and it gathers historical data to comprehend the preferences of a workload as PARIS. Hence, Scout is not affected by the model bootstrapping problem. However, in the selection of the next configuration to test, Scout uses an approach based on the Probability of Improvement (PI). Thus, it has a pure exploitative behavior that exposes it to a higher risk to get stuck in local optima [16]. The search process learns from previously evaluated configurations and from performance data of other workloads using transfer learning methods, thus decreasing the number of explorations required. However, the availability of offline knowledge is an undesirable assumption and it can be time consuming and expensive to build. The main advantage of running Scout is that it can largely reduce the cost of the exploration phase. However, exploring fewer configurations does not imply a small exploration cost because expensive configurations can be explored comparing, especially, when more and better configurations were explored. Furthermore, Scout does not use subsampling to reduce the cost spent.

Quasar. Quasar [23] predicts the amount of resources that should be allocated to a workload in order to meet the performance constraints. It aims at maximizing the resource utilization while meeting performance and QoS constraints for each workload. The amount of resources that are allocated, their types and possible interference between jobs running on the same physical machines may have impact on the performance of the job when it is run on that particular configuration. Hence, to maximize the performance of a job, Quasar uses a technique based on such as Collaborative Filtering (CF) [25, 75], to accelerate and to increase the accuracy of the predictions of the impact a given configuration might have on the overall performance of the job. Posteriorly, this information is used to allocate resources in order to determine the minimum amount of resources necessary to meet the constraints. However, CF requires offline knowledge and Quasar does not use subsampling

HCloud. A different system, called HCloud [22], is a hybrid provisioning system that determines if a job should allocate on-demand or reserved resources, leveraging on-demand resource for short-term resource needs and reserved resources for long-term jobs. For an incoming job, it collects information about resource preferences. Using this new knowledge, HCloud determines the type and quantity of resources that a job should use in order to satisfy the Quality of Service constraints. HCloud defines a policy to map jobs to resources. This policy ensures that reserved resources are utilized before on-demand resources. Applications that can be deployed on on-demand resources should not delay the scheduling of interference sensitive jobs. A dynamic adjustment of the utilization limits of the reserved instances should be done to reduce the queuing and to avoid performance deterioration. When a predefined limit for reserved resources usage is reached, the policy differentiates jobs by sensitivity to performance unpredictability, through the computation of resource quality, that dictates if a job should use on-demand or reserved resources. If reserved resources are selected and they are not available, the job enters in a queue until those become available. HCloud does not support constrains neither subsampling.

Proteus. Other approach aims to exploit the available transient revocable resources (like, Spot-Instances in AWS) in order to decrease the time and cost of distributed machine learning jobs, which divide the work by several workers controlled by a server. The optimization problem solve does not keep into account constraints. Proteus does not use subsampling in order to reduce the cost. This system, called Proteus [35], can combine both transient revocable and non-transient reliable machines and it handles the possibility of resources being revoked. By exploiting these two types of resources for running distributed machine learning jobs, Proteus can perform these jobs faster and/or cheaper.

2.5 Discussion

Table 2.2 provides a compact taxonomy of the works reviewed in Section 2.4, as well as of the systems developed in this dissertation, namely Nephele and Fabulinus (which will be presented in Chapter 3 and 4, respectively). Through the analysis of Table 2.2, it is possible to clarify not only the main differences between the reviewed state-of-the-art solutions but also clarify which gaps in the existing literature are being filled by the solutions presented next.

As we can see both Nephele and Fabulinus jointly optimize both cloud and application parameters. This is in common with Lynceus, which has first to shown the relevance of optimizing these two type of parameters in a joint fashion. Nephele and Fabulinus though consider different optimization problems than Lynceus and rely on subsampling. Nephele aims to identify the cheapest configuration that satisfies user defined constraints on accuracy and execution time. Unlike Lynceus, which assumes that ML jobs can only be trained on the full dataset, Nephele exploits the possibility of subsampling the dataset to reduce the job's cost, at the expense of a controlled degradation of the model's accuracy. As we will show in Chapter 6, thanks to the use of subsampling, Nephele can reduce the cost of the configuration that it recommends to be used in production (by a factor up to 4). Also, despite the inclusion of subsampling in the configuration space increases the cardinality of the search space, Nephele incurs exploration costs that are comparable, and often even lower, than Lynceus (despite Lynceus considers a relatively smaller search space by neglecting the possibility of using subsampled datasets). As we will see, in fact, the increase in the cardinality of the search space is compensated or even outweighed by the reduction of the cost for testing configurations using subsampled data set. Fabulinus targets a different optimization problem: maximizing accuracy using the full dataset subject to cost constraints, while testing configurations using subsampled datasets. Note that Fabolas addresses a similar optimization, except for two key differences: i) Fabolas does not support the definition of additional constraints on the configurations that it recommends; ii) Fabolas was designed and evaluated in the context of hyperparameter optimization and, unlike Fabuliuns, it does not account for the optimization of cloud-related parameters. Recall that Fabolas uses an acquisition function that exploits transfer learning techniques to extrapolate the knowledge on accuracy and cost gathered using subsampled datasets for the case in which the full dataset is used. In a nutshell, Fabulinus extends the acquisition function of Fabolas to account also for the likelihood that the cost constraints will be met by the configuration (using the full dataset) that will be recommended after having tested one more configuration using a subsampled dataset.

	Tunes application-level Parameters	Tunes Cloud-related Parameters	Base Optimization Techniques	Optimization Goal	Exploit Subsampling	Considers Additional Constrains
Hyperband	✓	✗	Model-free and Successive Halving	Minimize loss	✗	✗
BOHB	✓	✗	BO and Hyperband	Minimize loss	✗	✗
Fabolas	✓	✗	BO, GPs and Transfer Learning	Minimizes loss	✓	✗
Cherrypick	✗	✓	BO and GPs	Minimize cost	✗	Time
Lynceus	✓	✓	BO and Decision Tree	Minimize cost	✗	Accuracy and Time
Paris	✗	✓	Random forests	Maximize performance	✗	Cost
Scout	✗	✓	Pair-wise prediction	Maximize performance	✗	Cost
Quasar	✗	✓	Collaborative Filtering	Maximize resources utilization	✗	✓
Hcloud	✗	✓	Hybrid provisioning system	Determine type of resources (on-demand or reserved)	✗	✗
Proteus	✗	✓	Time-series based prediction	Minimize cost	✗	✗
Nephele	✓	✓	BO and Decision Tree	Minimize cost	✓	Accuracy and Time
Fabulinus	✓	✓	BO and GPs	Maximize accuracy	✓	Cost or Time

Table 2.2: Comparison between optimization goals and concerns of state-of-the-art and proposed systems (Nephele and Fabulinus)

3 Nephele

This chapter presents Nephele, a system to find optimal configurations to deploy machine learning jobs in the cloud. Nephele aims to find the best configuration to deploy a user-defined machine learning job in the cloud that minimizes the deployment cost subject to Quality of Service (QoS) constraints using subsampled datasets. According to Greek mythology, Nephele was the goddess of the clouds that had the power to control the clouds and the rains.

The optimization problem considered by Nephele is to find the configuration (including both cloud and model-related parameters) that minimizes the deployment cost to run a job in the cloud using subsampled datasets of size s, subject to constraints on both execution time and accuracy and a budget B for the exploration phase.

$$\begin{aligned} \underset{x,s}{\text{minimize}} \quad & C(x, s) \\ \text{subject to} \quad & A(x, s) \geq A_{\min} \\ & T(x, s) \leq T_{\max} \\ & \sum_{(x_k, s_k) \in \mathcal{S}} C(x_k, s_k) \leq B \end{aligned} \tag{3.1}$$

where C, A and T are the cost, accuracy and time functions, respectively, A_{min} and T_{max} are the accuracy and time constraints, x represents a configuration, s is the dataset size, and B is the budget for the exploration phase.

In the following sections, the implementation of Nephele is detailed. Firstly, Section 3.1 describes the design of the system. Then, Section 3.2 details the algorithm implemented by Nephele.

3.1 System Overview

Nephele is an online algorithm in the sense that it assumes no *a priori* knowledge on the job to be optimized or on previously optimized jobs. As such, in order to solve the optimization problem to find the optimal configuration, it is necessary to explore, at least partially, the configuration space. The optimal configuration is the one that minimizes the deployment cost and complies with the user-defined QoS constraints. These constraints, for example, can stipulate the maximum running time and/or the minimum performance required for a job. There is one additional constraint that specifies the amount of money that a user is willing to pay to explore the space to find the optimum.

Nephele leverages Bayesian Optimization (BO) techniques to solve the optimization problem and builds models for accuracy and cost. In order to improve these models and to find the optimal configuration, it needs to evaluate and deploy a configuration, thus, incurring a cost.

Nephele adds a new dimension to the search space that corresponds to the fraction of the dataset used as input to the ML job, given by $s = N_{sub}/N$, where N_{sub} and N are the size of the subsampled and the full datasets, respectively. Through the computation of the acquisition function, Nephele automatically determines the dataset size $s \in [0, 1]$ used to train a configuration x.

Reducing the dataset size using subsampling allows for reducing the computational demand for training the model, but it is also likely to degrade its accuracy. Nephele allows users to control this trade-off by letting them specify a constraint on the minimum accuracy that the model should achieve A_{min}).

In order to bootstrap the knowledge of the model, Nephele selects an initial set of configurations using Latin-Hypercube Sampling (LHS) [78]. Since the goal is to minimize deployment and exploration cost, Nephele uses a fixed initial budget B_{init} for the initial sampling, instead of sampling a percentage of configurations in the search space to construct the model, as CherryPick and Lynceus. Nephele uses a module call Sampler that randomly selects the configurations to sample. Then, it executes each configuration and collects the performance and cost measurements in order to construct the models.

After the initial sampling, Nephele constructs two models: a cost and an accuracy model. The Modeler is the module responsible for constructing and updating the two models built using the measurements gathered in each iteration. Then, these two models are used by a module called Selector that, through the computation of the acquisition function, chooses the next configuration to evaluate. Then, the selected configuration is executed. The Executor connects to the user-specified cloud provider and creates the cluster of VM. The size of the cluster and the VM's flavor is given by the configuration. The Executor deploys the job in the cloud on the selected configuration. At the end of execution, the Executor collects the necessary measurements and sends them to the Updater.

The Updater uses the information received to update the current state. It adds new information to the training set that contains the evaluated configurations and removes it from the set of unexplored configurations and decrements the current budget by the cost of the experiment. Then, the Updater informs the Modeler of the new measurements, who updates, retrains, and improves the models. Figure 3.1 represents the architectural scheme of Nephele.

3.2 Algorithm Description

This section describes the algorithm implemented by Nephele and the implementation details. The main concern of Nephele is to choose the next configuration to evaluate based on the current state (i.e., in the previous observations) in order to solve the optimization problem.

Algorithm 2 describes the system. Through the Sampler (Lines 7-10), Nephele uses LHS to select configurations to sample in order to bootstrap the accuracy and cost models. The initial sampling ends when the user-defined initial budget B_{init} ($B_{init} < B$) is spent. Each time a configuration is evaluated, the state is updated. The configuration pair (x, s) is removed from the set of unexplored configurations $\mathcal{T}$ and is added to the training set $\mathcal{S}$. The available budget is decremented by the cost of (x, s).

Therefore, the initial samples are used to build the accuracy and the cost models by the Modeler (Line 12). Nephele uses a bagging ensemble of Decision Trees (cf. Section 2.2.2) as a model. Posteriorly, the next configuration to evaluate is chosen (by the Selector (Line 13)). The selection of the next configuration to evaluate is described in Section 3.2.1.

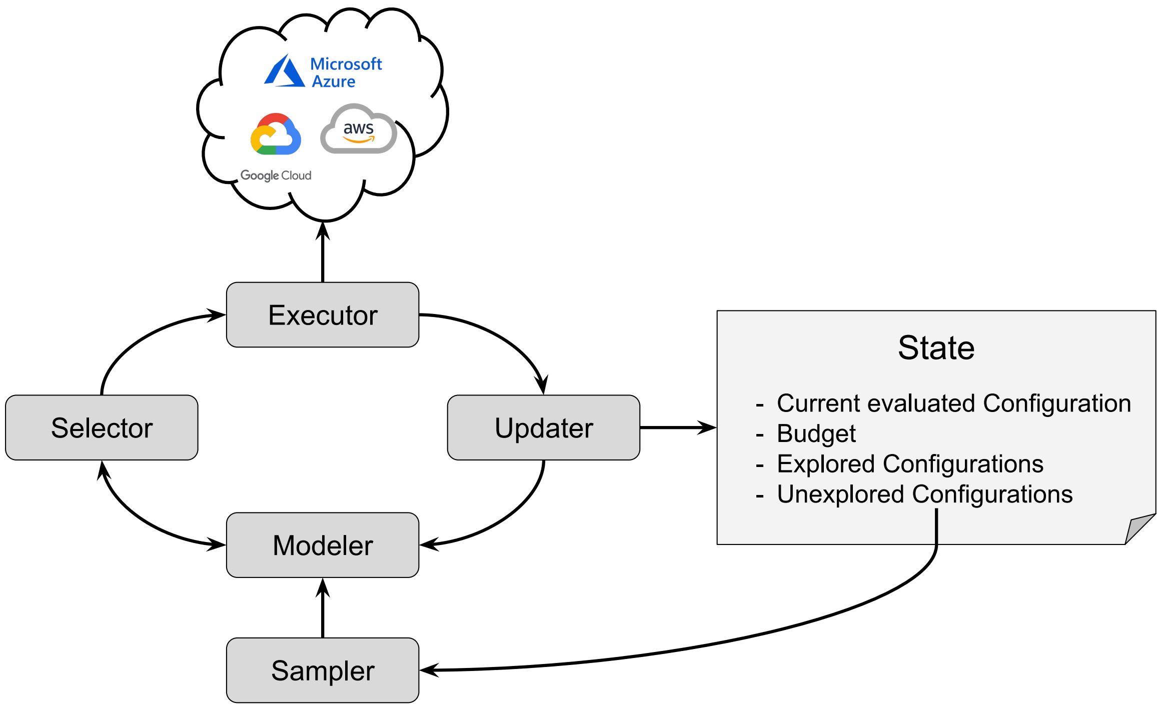

Microsoft Azure
Google Cloud
aws
Executor
Selector
Updater
Modeler
Sampler
State
- Current evaluated Configuration
- Budget
- Explored Configurations
- Unexplored Configurations

Figure 3.1: Nephele architecture

The chosen configurations are deployed in the cloud by the Executor (Lines 21-27) that creates the cluster of VMs and deploys the job in the cloud using the selected configuration. The cluster size and the VM flavor is defined in the configuration x to evaluate. During the deployment of the job in the cloud, the Executor periodically verifies if the available budget was spent.

When the job ends, the Updater (Lines 17-20) updates the training set with the accuracy and cost measurements, removes the evaluated configuration from the set of unexplored configurations and decrements the available budget. Then, a new iteration begins, the models are updated again, and this procedure is repeated.

When the available budget ends, it stops the execution and terminates all the machines in the cloud. Then, the exploration ends, and the final incumbent $(x_{inc}, s_{inc}) \in \mathcal{S}$ that minimizes the deployment cost and meets the QoS constraints is determined using all the previous gathered knowledge.

3.2.1 Selecting the Next Configuration

This Section describes and details the policies and approximations implemented by module Selector in Nephele in order to select the next configuration to evaluate.

The selection of the next configuration to sample in Nephele is based on the approach of Lynceus. In each iteration, the Selector chooses the next configuration to sample based on the accuracy and the cost model built. Since the search space can be very large (in the order of thousands of configurations), assess the quality of all the unexplored configurations can be costly, time-consuming, and intractable. Thus, a feasible set Γ is created in order to reduce the complexity of this problem (Algorithm 3, Line 2). This feasible set contains the configurations that are predicted to comply with the constraints, i.e., configurations predicted to have higher accuracy than the accuracy constraint A_{min}, lower execution time than the time constraint T_{max} and a deployment cost smaller than the available budget β. In order to

Algorithm 2 Nephele

```
1:  function MAIN($B$, $B_{init}$, $h$)
    ▷ $B$: Budget, $B_{init}$: Budget for the initial sampling, $h$: maximum horizon
2:      $\mathcal{S} \leftarrow \emptyset$                                          ▷ Training set
3:      $\mathcal{T} \leftarrow$ Whole search space                      ▷ Set of untested configurations
4:      $\mathcal{D} \leftarrow$ All possible dataset sizes              ▷ Set with the possible dataset sizes
5:      $\beta \leftarrow B$                                                        ▷ Current budget
6:      $\eta \leftarrow B_{init}$                                                  ▷ Budget for the initial sampling
7:      while $\eta \geq 0$ do                                                      ▷ Bootstrap
8:          $(x, s, a, c) \leftarrow LHS(\mathcal{T})$                              ▷ Initial sampling
9:          UPDATE$(x, s, a, c)$                                                    ▷ Updater
10:         $\eta \leftarrow \eta - c$
11:     while $\beta \geq 0$ do
12:         Fit Decision Trees model for $A(x, s)$ and $C(x, s)$ using $(x, s) \in \mathcal{S}$   ▷ Modeler
13:         $(x, s) \leftarrow$ NEXTCONFIG $(h)$                                    ▷ Selector
14:         $(a, c) \leftarrow$ EVALUATE $(x, s)$                                   ▷ Executor
15:         UPDATE$(x, s, a, c)$
16:     return $\arg\min_{(x,s)\in\mathcal{S}} C(x, s)$, s.t. $A(x, s) \geq A_{\min} \wedge T(x, s) \leq T_{\max}$
        ▷ Select the incumbent configuration $(x_{inc}, s_{inc})$

17: function UPDATE$(x, s, a, c)$
18:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{x, s, a, c\}$                   ▷ Update the training set
19:     $\mathcal{T} \leftarrow \mathcal{T} \setminus (x, s)$      ▷ Remove $(x, s)$ from the set of untested configurations
20:     $\beta \leftarrow \beta - c$                                                ▷ Decrement budget

21: function EVALUATE$(x, s)$
22:     Deploy the job in configuration $x$ and using the dataset size $s$
23:     while $(a, c) \leftarrow$ Run$(x, s)$ do        ▷ Runs the jobs in the selected configuration $(x, s)$
24:         Check available budget
25:         if $\beta \leq 0$ then                       ▷ Verifies if the available budget was spent
26:             Stop evaluation
27:     return $(a, c)$
```

avoid mispredictions due to the high uncertainty of the models, Nephele adds to the feasible set the configurations whose predicted mean values of accuracy and cost comply with the constraints. The execution time of a configuration (x, s) is calculated through the division between the prediction given by the cost model $C(x, s)$ and the cost per unit of time of the allocated cloud resources given in the configuration $P(x, s)$, i.e., $T(x, s) = C(x, s) \cdot P(x, s)$.

If the feasible set Γ is empty, the next configuration is randomly selected among the configurations in the unexplored set $\mathcal{T}$. If the feasible set Γ is not empty, Nephele uses lookahead to simulate the path of configurations starting at $(x, s) \in \Gamma$ that could be created if the configuration (x, s) was evaluated in the next iteration. Each simulated path has a maximum exploration horizon h defined by the user.

When the user-defined maximum lookahead horizon h is zero, which means that only a single-step-ahead is simulated, the reward function R is computed for all configurations in Γ. In this particular case, the reward function is given by the computation of the Constrained Expected Improvement (EI$_C$) (Equation 2.17) that coincides with the acquisition function (cf. Section 2.3.1.1) used. The next configuration to evaluate is the one that maximizes the ratio between the reward and the deployment cost, i.e.,

$$(x_{next}, s_{next}) = \arg\max_{(x,s)\in\Gamma} \frac{R(x, s)}{C(x, s)} \tag{3.2}$$

When the maximum lookahead horizon h is not zero, Nephele simulates the possible path created by the evaluation of $(x, s) \in \Gamma$. Therefore, Nephele tries to evaluate not only the cost of trying a configuration but also the impact and the contribution in the long-term brought by that experiment to improve the models and to find the optimal configuration. It evaluates the contribution brought by a configuration (x, s) and also by the following h simulated iterations. For that, it calculates the expected reward of a simulated path.

However, the exploration of all the possible paths considering all the unexplored configurations creates an intractable, expensive, and time-consuming problem. In order to reduce the complexity and to avoid exploring all the possible paths, in each lookahead step is determined a new feasible set Ω using the same condition of the previous feasible set Γ. Then, for each configuration in Ω, it is computed the EI_C. In order to further reduce the search complexity, the next configuration simulated in the path is the one in Ω that maximizes the EI_C. Thus, only one path per configuration in Γ is built.

Based on Lynceus, it was implemented a recursive algorithm that simulates the paths. In each lookahead step, the current state is cloned in order to be possible to return to the original state. The utility and the cost are estimated for each configuration in the path. This process is repeated h times until the utility and the cost of the last configuration in the path is calculated. Then, the reward and the total cost of the path are computed by adding the estimated utilities and costs of all configurations in the path, respectively.

The estimation of the expected utilities using lookahead requires the computation of nested expectations, which do not have a closed-form expression. Thus, in order to compute the expected utilities of configurations in the path, these expectations are approximated using Gauss-Hermite (G-H) quadrature [55]. This quadrature approximates integrals through the sampling of N points. Each sampled point has an associated weight that represents the likelihood of the sampled value being the real one. This way, each weight determines the contribution of the sampled value to the estimation of the path utility.

In order to decrease the complexity of computing the G-H quadrature, only 3 points for each prediction are sampled. Since Nephele uses the accuracy and the cost model to predict the next configurations in the path, it is necessary to approximate the accuracy and the cost predictions, which follow a Normal distribution, using a bivariative Gauss-Hermite quadrature [41]. The reward and the cost functions are approximated by

$$\sum_{i=1}^{N}\sum_{j=1}^{M} w_i w_j f(c_i, a_j), \tag{3.3}$$

where f can be the reward or the cost function, c_i and a_j are the sampled points of the cost and accuracy predictions, respectively, w_i and w_j are the associated weights of a Gauss-Hermite quadrature. Thus, $N \times M$ pairs of accuracy and cost points (c_i, a_j, w_i, w_j) are obtained. For each possible prediction (c_i, a_j) of a configuration (x, s), the models are updated (Algorithm 3, Lines 23) and the next configuration in the path is determined (Algorithm 3, Lines 30-32). In order to decrease the computational complexity of computing a bivariative G-H quadrature, it is possible to approximate the cost prediction through a G-H quadrature and use the mean value of the accuracy prediction (in other words, M=1) to update the accuracy model. This way, only N pairs (c_i, μ_a, w_i) are obtained and computed.

Parameter γ (Algorithm 3, Lines 27-28) is a discount factor that enables the possibility to attribute different contributions to configurations in the path based on the depth level (i.e., lookahead step) in which the prediction is done. Thus, when $\gamma \in]0, 1[$, the contribution of the estimation decreases with the increase of the depth level. When $\gamma = 0$, the contribution of the path is not taken into account, and the

Algorithm 3 Selection of the next configuration

```
1:  function NEXTCONFIG($h$)
2:      $\Gamma \leftarrow \{x \in \mathcal{T} \wedge s \in \mathcal{D} : C(x, s|\mathcal{S}) \leq \beta \wedge A(x, s|\mathcal{S}) \geq A_{\min} \wedge T(x, s|\mathcal{S}) \leq T_{\max}\}$    ▷ Feasible set
3:      if $\Gamma == \emptyset$ then
4:          $(x_{next}, s_{next}) \leftarrow$ Random($\mathcal{T}$, $\mathcal{D}$)    ▷ Selects a configuration at random
5:      else
6:          for $(x_k, s_k) \in \Gamma$ do    ▷ Compute rewards of exploration paths that start with any $x \in \Gamma$
7:              $(R_k, C_k) =$ EXPLOREPATHS($x_k, s_k, h, \mathcal{S}, \mathcal{T}, \beta$)
8:          $(x_{next}, s_{next}) = \arg\max_{(x,s)\in\Gamma} R(x, s)/C(x, s)$    ▷ Select configuration that maximizes reward/cost
9:      return $(x_{next}, s_{next})$

10: function EXPLOREPATHS($x$, $s$, $h$, $\mathcal{S}$, $\mathcal{T}$, $\beta$)
11:     $R \leftarrow EI_c(x, s)$    ▷ Calculate the $EI_c$ of the first configuration in the path
12:     $C \leftarrow C(x, s)$    ▷ Predicts the cost of the first configuration in the path
13:     if $h == 0$ then    ▷ Maximum Lookahead horizon
14:         return $(R, C)$    ▷ Return the reward and cost
15:     else
16:         $\langle c_i, w_i \rangle \leftarrow$ G-H$(C(x, s)), i = 1, \ldots, N$    ▷ Gauss-Hermite quadrature
17:         $\langle a_j, w_j \rangle \leftarrow$ G-H$(A(x, s)), j = 1, \ldots, M$
18:         for $(i = 1, \ldots, N)$ do    ▷ Create state with speculated values
19:             for $(j = 1, \ldots, M)$ do
20:                 $\mathcal{S}' \leftarrow \mathcal{S} \cup (x, s, c_i, a_j)$
21:                 $\beta' \leftarrow \beta - c_i$
22:                 $\mathcal{T}' \leftarrow \mathcal{T} \setminus (x, s)$
23:                 Retrain models $A'(x, s)$ and $C'(x, s)$ using $(x, s) \in \mathcal{S}'$
                    ▷ Update the accuracy and cost model using the predicted values
24:                 $(x', s') \leftarrow$ NEXTSTEP($\mathcal{T}'$, $\mathcal{S}'$)    ▷ Select next configuration $x'$ using the updated model
25:                 if $(x' == null)$ then Continue    ▷ All predicted configurations are infeasible
26:                 $(r, c) \leftarrow$ EXPLOREPATHS($x', s', h - 1, \mathcal{S}', \mathcal{T}', \beta'$)
                    ▷ Compute reward and cost of sub-path of length $h - 1$ rooted in $x'$
27:                 $C \leftarrow C + \gamma w_i w_j c$
28:                 $R \leftarrow R + \gamma w_i w_j r$    ▷ Calculates the reward and cost of a path weighted by $w_i w_j$
29:         return $(R, C)$

30: function NEXTSTEP($\mathcal{T}$, $\mathcal{S}$)    ▷ Select configuration that maximizes $EI_c$ and meet the constraints
31:     $\Omega \leftarrow \{x \in \mathcal{T} \wedge s \in \mathcal{D} : C(x, s|\mathcal{S}) \leq \beta \wedge A(x, s|\mathcal{S}) \geq A_{\min} \wedge T(x, s|\mathcal{S}) \leq T_{\max}\}$
32:     if $\Omega == \emptyset$ then return (null) else return $\arg\max_{(x,s)\in\Omega} EI_C(x, s)$
```

algorithm will present the same behavior as when no lookahead is used ($h = 0$). When $\gamma = 1$, the weight attributed does not depend on the depth level, and all the predicted utilities offer the same contribution for the path.

Main differences with respect to Lynceus As mentioned the logic employed by Lynceus and Nephele to select the next configuration share several commonalities (e.g., they both use lookahead techniques and consider constraints on budget and execution time). In the following, we clarify what are the key differences between Nephele and Lynceus:

- Subsampling can degrade the accuracy of the resulting models. As such, Nephele needs to predict the impact of the configuration's choice on both execution time and accuracy.

- Nephele considers a larger search space that includes the subsampling rate as an additional dimension.

4 Fabulinus

This chapter introduces Fabulinus (Fast Bayesian Optimization of Constrained Machine Learning Cloud Jobs), a system to optimize machine learning (ML) jobs in the cloud in order to maximize the performance subject to user-defined constraints. The name Fabulinus in Roman mythology was the education god, who taught the children to speak and increase their knowledge.

Similar to Nephele presented in Chapter 3, Fabulinus aims to find the optimal configuration to deploy a ML job in the cloud. Therefore, it tries to optimize the allocation of cloud resources and tune specific application parameters. However, Fabulinus solves a different optimization problem than Nephele.

Fabulinus aims to maximize the accuracy of a ML model trained over the full dataset, subject to constraints on the maximum cost for running the job in the cloud. In order to enhance the efficiency of the optimization process, though, Fabulinus only tests configurations by training the model using subsampled datasets. More formally, Fabulinus consider the following optimization problem:

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & A(x, s=1) \\ \text{subject to} \quad & C(x, s=1) \leq C_{\max}, \end{aligned} \tag{4.1}$$

where A and C are the accuracy and the cost functions, respectively, x is a configuration, s is the dataset size, and C_{max} is a cost constraint. The constraint specified serves to limit the cost that the user has to pay to deploy the job. However, instead of defining a cost constraint, the user could define a time constraint (since cost and time are not independent, the definition of both constraints at the same time would create an additional difficulty).

Section 4.1 gives a description of the proposed system and Section 4.2 details the algorithm implemented by Fabulinus.

4.1 System Overview

Fabulinus is a system for optimizing the cloud resources and hyperparameters of machine learning applications deployed in the cloud, in order to maximize the accuracy of the job subject to a user-defined cost constraint that specifies how much a user is willing to pay for a given job. Fabulinus uses subsampling techniques in order to evaluate cheaper and faster configurations, but it tries to maximize the accuracy on the full dataset. This way, the exploration cost can be reduced. It leverages Bayesian Optimization to model the accuracy and the cost functions, adding a dimension to the search space that corresponds to the dataset size s.

While the goal is to maximize the accuracy using the full dataset, Fabulinus evaluates the accuracy and the cost functions on subsampled datasets, which are usually cheaper. Then, using transfer learning techniques, it predicts the values of the objective and cost functions for the full dataset. The additional dimension of the search space corresponds to the fraction of the dataset size used on the job, which is given by $s = N_{sub}/N$, where N_{sub} and N are the size of the subsampled and the full datasets, respectively.

The correlation across the dataset size is unknown *a priori*. So, in order to tackle this limitation, Fabulinus' strategy consists of selecting configurations that will increase the knowledge about the scaling behavior and about the optimal configuration that maximizes the accuracy and meets the constraints on the full dataset.

Fabulinus builds two models: an accuracy and a cost model. It uses GPs to construct each model, and the kernel used is a factorized kernel given by the multiplication between the Matérn 5/2 kernel and finite-rank covariance function in s, as proposed in Fabolas [48]. The configuration pair (x, s) to evaluate is selected using an acquisition function that trades-off the information gain about the optimum for $s = 1$, the cost of evaluating (x, s), and the probability that the predicted incumbent complies with the constraint.

The architecture of Fabulinus is similar to Nephele. It is composed of four modules: a Sampler for initial sampling, a Modeler to construct, update and improve the models, an Updater to update the current state, a Selector to select the configuration to evaluate next, and the Executor to deploy the job in the cloud given a configuration.

4.2 Algorithm Description

This section details the algorithm of Fabulinus and its implementation. Algorithm 4 provides a high level overview of Fabulinus Logic.

The system builds an initial model for both accuracy and cost functions using LHS (Algorithm 4, Lines 4-6), evaluating a user-defined number of configurations. Each configuration x is evaluated on all the possible subsampled dataset sizes in order to provide better information on the scaling behavior. Then, the state is updated and the accuracy and cost models are built using these initial evaluations (Line 8).

The next configuration (x, s) to be tested is then selected using an acquisition function that will be detailed in Section 4.2.1. Then, the selected configuration (x, s) is evaluated and, after the evaluation, the state is updated, the accuracy and cost of (x, s) are added to the training set, and the configuration is removed from the set of unexplored configurations. The incumbent (i.e., the configuration deemed to be the optimal in the present moment) is the configuration that maximizes the predicted accuracy and meets the constraint with a probability higher than 99%. At the end of each iteration, the incumbent is determined based on the predicted accuracy and cost for all possible configurations that use the full dataset $(s = 1)$. Fabulinus stops the optimization process when it reaches a user-defined number of iterations. In the end, it predicts the final incumbent that solves the optimization problem.

It should be noted that the Algorithm 4 is the same used by Fabolas with the following exceptions:
- Fabolas does not include in the configuration space any cloud-related parameter;
- Fabolas does not account for constraints on the maximum cost of running the job in the cloud. As a consequence, Fabolas uses a different acquisition function and a different logic for determining the incumbent.

Algorithm 4 Fabulinus

```
1:  function MAIN($N$, $M$, $\mathcal{D}$)
        ▷ $N$: Number of configurations for LHS, $M$: Number of iterations, $\mathcal{D}$: possible dataset sizes
2:      $\mathcal{S} \leftarrow \emptyset$                                   ▷ Training set
3:      $\mathcal{T} \leftarrow$ Whole search space                          ▷ Set of untested configurations
4:      for $(i = 1, \dots, N)$ do                                           ▷ Bootstrap
5:          $(x, s, a, c) \leftarrow LHS(\mathcal{T})$                       ▷ Initial sampling
6:          UPDATE$(x, s, a, c)$
7:      for $(i = N, \dots, M)$ do
8:          Fit GPs model for $A(x, s)$ and $C(x, s)$ using $(x, s) \in \mathcal{S}$
9:          $(x, s) \leftarrow$ NEXTCONFIG ()
10:         $(a, c) \leftarrow$ Evaluate $(x, s)$                            ▷ Test configuration $s$ using the dataset size $s$
11:         UPDATE$(x, s, a, c)$
12:         Choose incumbent $x_{inc}$ based on the predicted accuracy and cost at $s = 1$

13: function UPDATE$(x, s, a, c)$
14:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{x, s, a, c\}$             ▷ Update the training set
15:     $\mathcal{T} \leftarrow \mathcal{T} \setminus (x, s)$                ▷ Remove $(x, s)$ from the set of untested configurations
```

4.2.1 Selecting the Next Configuration

The acquisition function used by Fabulinus extends the one proposed in Fabolas, which uses transfer learning techniques (Multi-Task Bayesian Optimization (MTBO) [81]) to predict the optimal configuration on the full dataset using the knowledge gained through the evaluation of subsampled datasets.

In order to select the next configuration and the dataset size to evaluate, Fabolas extends an acquisition function called Entropy Search (ES) given by Equation 4.3 that maximizes the information gain per unit cost about the distribution of the optimum on the full dataset, i.e.,

$$p_{max}^{s=1}(x|\mathcal{S} \cup \{(x,s,y)\}) = p(x \in arg \max_{x' \in \mathbb{X}} A(x', s=1)|\mathcal{S} \cup \{(x,s,y)\}) \tag{4.2}$$

The acquisition function is normalized with respect to the total cost, i.e., the deployment cost $C(x, s)$ plus the overhead cost $C_{overhead}$, in order to trade-off the information gain and cost.

$$a_F(x,s) = \frac{1}{C(x,s) + C_{overhead}} \mathbb{E}_{p(y|x,s,\mathcal{S})} \left[\int p_{max}^{s=1}(x'|\mathcal{S} \cup \{(x,s,y)\}) \cdot \log \frac{p_{max}^{s=1}(x'|\mathcal{S} \cup \{(x,s,y)\})}{u(x')} dx' \right] \tag{4.3}$$

Fabulinus extended the acquisition function proposed in Fabolas to keep into account constraints. The new acquisition function, called constrained Entropy Search (ES$_\mathrm{C}$), selects the configuration that maximizes the information gain per unit cost about the optimum at $s = 1$ and the probability that the predicted incumbent complies with the constraint but only evaluates configurations on subsampled datasets. It computes the relative entropy between $p_{max}^{s=1}(x'|\mathcal{S} \cup \{(x,s,y)\})$ and a uniform distribution $u(x')$, with expectations taken over the evaluation y to be obtained at x, times the probability of the constraint being met. The constrained Entropy Search (ES$_\mathrm{C}$) is given by

$$ES_C(x,s) = a_F(x,s) \cdot CEA(x,s). \tag{4.4}$$

$a_F(x, s)$ is the acquisition function used by Fabolas (Equation 4.3). Roughly speaking, the Constrained Expected Accuracy (CEA) captures the likelihood that the incumbent output by the model at the next exploration step, i.e., after testing (x, s), will meet the constraints. Note that the incumbent at the next

Algorithm 5 Selection of the next configuration

```
1: function NEXTCONFIG
2:     $\Gamma \leftarrow$ Creation of the set containing the configurations to compute the acquisition function
3:     for $x \in \Gamma$ do
4:         for $s \in \mathcal{D}$ do
5:             Clone the accuracy and cost models and the training set $\mathcal{S}$
6:             $a' \leftarrow A'(x, s)$
7:             $c' \leftarrow C'(x, s)$
8:             $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{x, s, a', c'\}$
9:             Fit GPs model for $A'(x', s')$ and $C'(x', s')$ using $(x', s') \in \mathcal{S}'$
10:            Compute the acquisition function $ES_C(x, s)$          ▷ Compute Equation 4.4
11:    return $\arg\max_{x \in \Gamma} ES_C(x, s)$     ▷ Select the configuration that maximizes the acquisition fucntion
```

iteration, differs, in general, from the current incumbent, given that the knowledge of the model will be updated with information on the cost and accuracy of (x, s) (in case (x, s) is actually selected by the acquisition function). Thus, using the current model to predict the incumbent at the next step is likely to produce poor results. We solve this problem by emulating the sampling of (x, s) via the cost and accuracy models obtaining $A(x, s)$ and $C(x, s)$, respectively. We then train a new model on a dataset composed by the current dataset (i.e., the set of configurations tested so far) and the tuple $(x, s, A(x, s), C(x, s))$. More precisely, denoting with x_{inc}, the incumbent at the next iteration, the CEA is defined by the product between the predicted accuracy for x_{inc} and the probability that the cost of x_{inc} satisfies the cost constraint, after having trained a model over a dataset extended with $(x, s, A(x, s), C(x, s))$.

$$CEA(x, s) = A(x_{inc}, s = 1|\mathcal{S} \cup (x, s, A(x, s), C(x, s))) \cdot P(C(x_{inc}, s = 1|\mathcal{S} \cup (x, s, A(x, s), C(x, s))) \leq C_{max}) \tag{4.5}$$

4.2.2 Implementation Details

In order to speed up the computation of the acquisition function (Equation 4.4), it is created a set Γ, which contains the configurations to compute the acquisition function. Fabulinus chooses the top β configurations that maximize the CEA. This way, the acquisition function is only computed with configurations whose predicted incumbent has high accuracy and a high probability that the incumbent cost complies with the constraint.

To compute Equation 4.4, the models are cloned to save the current state and for each configuration $x \in \Gamma$ and for all the possible values of s, the models are retrained using the predicted accuracy and cost values for (x, s). Using the updated models with (x, s), the acquisition function is computed. The configuration (x, s) that maximizes the acquisition function of Equation 4.4 is evaluated next.

5 Datasets

This chapter provides a description of the machine learning jobs deployed and the respective datasets gathered for this particular work in order to evaluate the solutions proposed in Chapter 3 and Chapter 4, namely Nephele and Fabulinus. Three different ML jobs, corresponding to the distributed training of three different NNs, were deployed in the cloud Each job was run in AWS EC2 for all possible configurations and dataset sizes.

The neural networks were implemented resorting to the Tensorflow framework [1] in Python. The dataset used for training the NNs is the MNIST database [24] (a standard benchmark for evaluating NNs), that contains 70000 images of 28x28 pixels of handwritten digits, 60000 images for training and 10000 images for testing. Supervised methods were used to train the networks, i.e., the classification process is learnt from input-output pairs.

The three different NNs implemented have different architectures, which are explained in the following sections.

5.1 Convolutional Neural Network (CNN)

This first dataset was obtained by training a convolutional neural network using the MNIST database. This network is often used for image recognition problems. It is composed of an input and an output layer, two convolutional layers, which consist of a series of convolution layers employing a series of convolution operations, and two pooling layers. The key idea is based on a feed-forward network, that extracts small features from the previous layer and, through convolution and pooling operations, these features are given as an input to the next layers. This way, CNN permits the extraction of features of an image and the conversion into a lower dimension without losing its characteristics. The last two layers are fully connected. The output layer is a fully connected layer composed of 10 neurons and classifies the input, assigning a probability of each class given the input. The architectural scheme of the implemented CNN is represented in Figure 5.1.

5.2 Multilayer Neural Network

The second dataset was collected through the training of a multilayer perceptron [20] composed by one input and output layers, and two fully connected hidden layers with 256 neurons each. The input has 784 features (corresponding to the pixels of an image). The output layer has 10 neurons and classifies an input in one of the 10 different classes that correspond to digits between 0 to 9. The activation function of each neuron is the Softmax function [31], which defines the response of a neuron given and input or a set of inputs. Each neuron is connected to all the neurons of the next layer (i.e., it is a fully connected network) and the edge between two neurons has an associated weight that is

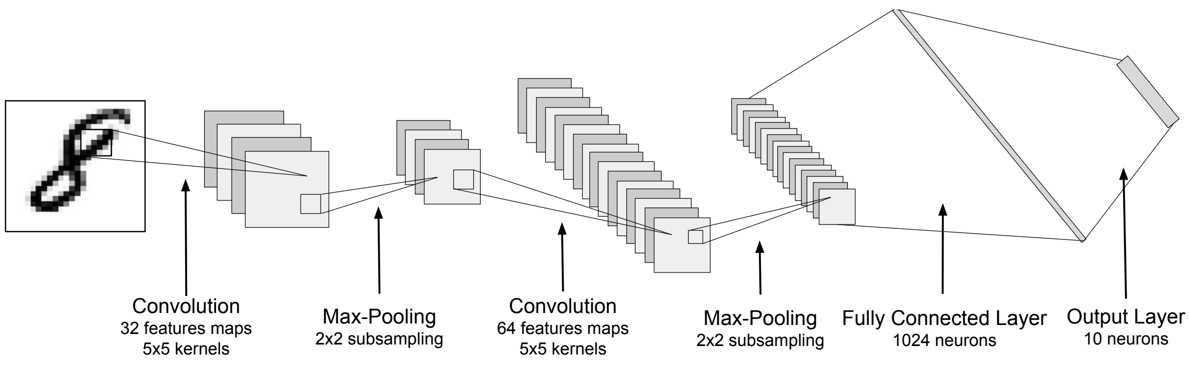


Figure 5.1: CNN architecture

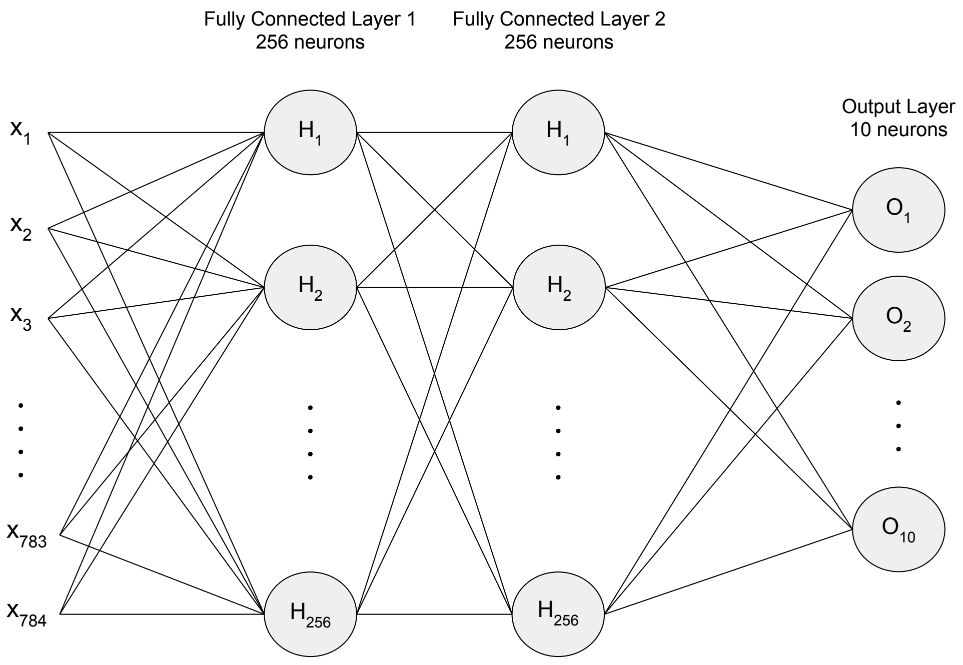


Figure 5.2: Multilayer NN architecture

calculated using a supervised learning technique designated backpropagation. This technique aims at minimizing the chosen loss function, which in this case is the cross entropy. Given an input, the basic idea is to calculate in the forward direction the output of each layer. In the output layer, the input is classified. During training, desired values are known and, therefore, the weights can be adjusted in order to reduce the loss function. The weights are calculated using the Adam Optimization algorithm [47]. The architecture of this network is shown in Figure 5.2.

5.3 Recurrent Neural Network (RNN)

The last dataset was obtained training a recurrent neural network. Using a RNN, the connection between neurons forms a direct graph along a temporal sequence and, therefore, it permits feedback loops. This way, it uses the internal state (or memory because it depends on past events) to process sequences of inputs and, thus, exhibits a temporal dynamic behavior. The knowledge learnt from previous experiments influences the decisions.

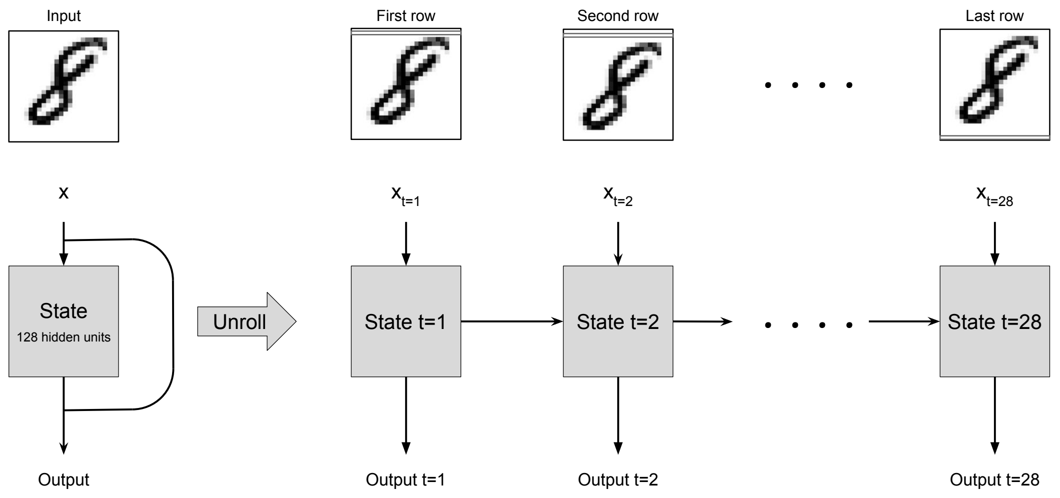


Figure 5.3: RNN architecture implemented by a LSTM network

The RNN implemented is a Long Short-Term Memory (LSTM) network [15]. A LSTM network can add and remove information from a cell state, carefully regulated by structures called gates and, thus, the network selects the information to retain adjusting the flow of information in and out of a cell. Each cell is composed of a set of neurons that represents the capacity of the NN. The scheme of the RNN implemented is represented in Figure 5.3.

5.4 Training the Neural Networks

Each network was trained with the full MNIST dataset and with four different subsampled datasets. These subsampled datasets were obtained by removing a predefined number of images and the respective labels from the full dataset. In order to check the accuracy of the network under training, a batch containing 5000 images is created from the MNIST test set. This accuracy check is performed by an extra worker, designated *Bookkeeper*, that verifies and saves the evolution of performance over time and checks stopping criteria.

Two stopping conditions were defined to bring the training of the NNs to an end: i) having observed during the training process every instance of the dataset with probability greater than α (which we set to 0.9), and ii) the maximum training time is lower than T_{max} (which we set to 10 minutes).

The first stop condition ensures that larger subsampling rates will require longer runs, while ensuring ”fair” conditions for learning independently (same probability of observing an instance of the dataset) of the chosen subsampling rate. Additional details on how this stop condition is implemented in the *Parameter Server* model are provided in Appendix A. The second stop condition serves the pragmatical purpose of limiting the maximum latency and monetary cost for collecting the datasets.

For each subsampled dataset, there are 288 possible configurations. Therefore, each dataset for each network has a total of 1440 configurations. In order to decrease the uncertainty and to avoid outliers concerning the metrics gathered during training, each configuration was run three times for each NN and, posteriorly, the average and the standard deviation of the results obtained in the three runs were calculated.

Cloud Parameters		NN's Parameters			Dataset Size
VM Flavor	No. of Cores	Learning Rate	Batch Size	Traning Mode	s
	- 8				- 0.0167
- t2.small	- 16	- 10^{-3}			- 0.1
- t2.medium	- 32	- 10^{-3}	- 16	- synchronous	- 0.25
- t2.xlarge	- 48	- 10^{-5}	- 256	- asynchronous	- 0.5
- t2.2xlarge	- 64				- 1
	- 80				

Table 5.1: Possible values for each parameter of a configuration

Each configuration is composed of a combination of cloud resources (in particular VMs' type/flavor and cluster size, i.e., number of machines) and internal hyperparameters of neural network applications (learning rate, batch size, and training mode) and the percentage of dataset size used to train. In order to use small datasets, it was sampled 1.67%, 10%, 25%, 50% of data from the full dataset, creating new datasets with 1000 ($s = 0.0167$), 6000 ($s = 0.1$), 15000 ($s = 0.25$) and 30000 ($s = 0.5$) images for training, respectively.

Table 5.1 shows the possible values for the different parameters that compose a configuration. The considered configurations encompass VMs of four different sizes and of the t2 family on AWS EC2. When the number of cores required in a configuration exceeds the number of cores of a single VM of a given type, the job is deployed in multiple VMs of the same type. The application specific parameters include the number of images used for training in each iteration (batch size), the learning rate of the optimization algorithm (Stochastic Gradient Descent [74, 46, 10]) used to calculate the weights in VMs and the training mode (synchronous or asynchronous).

The scripts to deploy, to train the VMs, and to collect the datasets were written in Python. Given a configuration, the Executor creates the cluster of machines in the cloud and launches the scripts to execute the user-specified application. It deploys a given job in a selected configuration. The Executor runs in a single machine (a VM of flavor t2.micro) and acts as job coordinator. These scripts to implement the Executor were written in Python using the Boto3 library, an AWS SDK for Python. Each configuration requires two additional VMs than the number specified by the configuration, one to be the *Parameter Server* and the other to be *Bookkeeper*. The *Parameter Server* coordinates the training process, keeps records of the VM parameters, and calculates the new weights to minimize the loss function. The *Parameter Server* always runs on a VM of flavor t2.2xlarge in order to reduce the latency of messages and to speed up the computation of the weights. The *Bookkeeper* measures the model's accuracy at predefined time intervals of 5 seconds to store the evolution of the performance over time. The *Bookkeeper* runs in a separate machine to be completely independent of the workers. This way, it does not affect the training process. It has the same VM flavor as the other workers.

6 Nephele Evaluation

This chapter presents the experiments performed in order to evaluate Nephele (see Chapter 3). Firstly, Section 6.1 details the baselines and metrics used for comparing Nephele. Section 6.2 details the implementation and the selection of the inputs and settings for running Nephele. Section 6.3 studies the impact of subsampling on cost in production, i.e., evaluates if it is possible to use subsampling in order to reduce the deployment cost of running a machine learning job in the cloud. This section also evaluates the impact of subsampling on the lookahead technique. Then, Section 6.4 evaluates the use of subsampling in order to reduce the exploration cost to find the optimum.

6.1 Evaluation Setup

This section details and explains the selection of state-of-the-art systems with similar optimization goals that aim to optimize cloud resources and specific application parameters with which Nephele is compared. Then, the metrics used to evaluate and compare the systems are described.

Baselines for Comparison. Nephele is compared with Lynceus [19] and CherryPick [2]. These two systems aim to optimize the allocation of resources in the cloud in order to minimize the deployment cost subject to user-defined QoS constrains. Furthermore, Lynceus also tries to minimize the cost of the exploration phase and incorporates a budget for the exploration phase and extends CherryPick because it allows the tuning of specific application parameters. Lynceus also use lookahead technique in order to reduce the exploration cost. Both systems use BO to build a cost model.

In the original version, CherryPick does not take into account the exploration cost and the budget, and however, for a fairer comparison, CherryPick was extended to incorporate a budget. Also, both Lynceus and CherryPick use a percentage of the search space for the initial sampling to bootstrap the model. Since Nephele uses an initial budget for the initial sampling, Lynceus and Cherrypick were also implemented to use an initial budget for a fairer comparison. However, these two systems do not use subsampling, unlike Nephele. These three systems are also compared with a Random selection of the next configuration to evaluate. To evaluate the various systems fairly, we consider the same stop condition: stop when the budget is over.

Evaluation Metrics. To compare and evaluate Nephele with the other two systems, the metric used is the Cost Normalized with respect to the Optimum (CNO). The CNO quantifies the quality of the incumbent and is given by the deployment cost of a configuration x normalized to the cost of the optimal configuration x_{opt}, i.e.,

$$CNO(x) = \frac{C(x)}{C(x_{opt})}. \tag{6.1}$$

The lower the CNO, the better the configuration. The optimal configuration has a CNO of 1. The CNO is evaluated as a function of the exploration number, as well as cost and time of the optimization process.

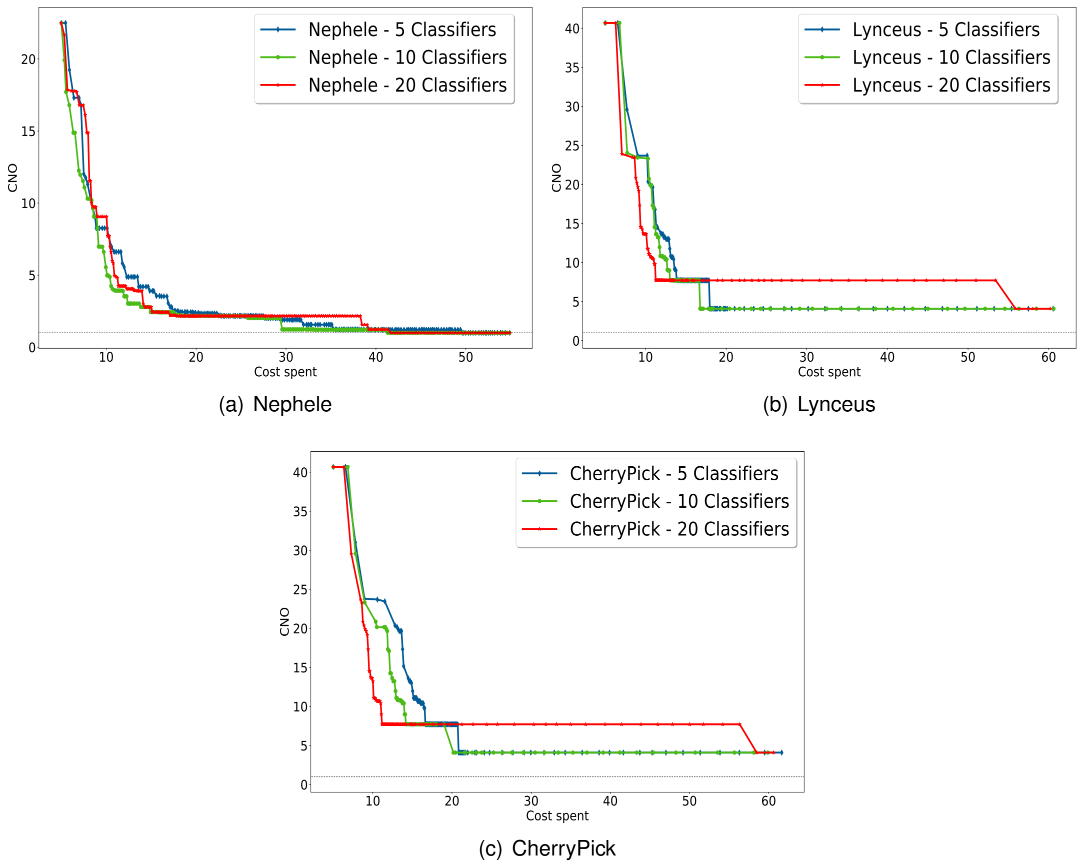


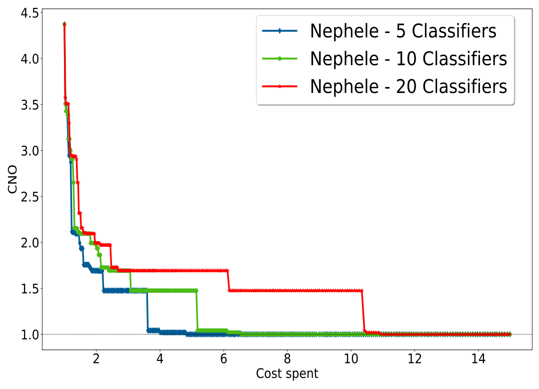
Figure 6.1: Comparison between different number of classifiers in the ensemble to train a CNN

6.2 System Implementation and Experimental Setup

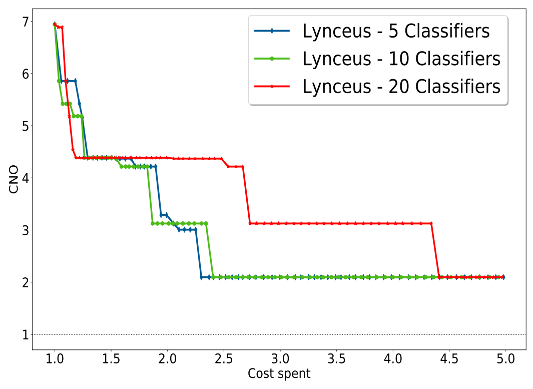
This section details the system implementation and the settings to run the experiments in order to evaluate Nephele.

System Implementation. Nephele was implemented in Java. In order to build the accuracy and cost models, Nephele uses an ensemble of decision trees (cf. Section 2.2.2). The ensemble of decision trees was implemented using the random tree algorithm of the Weka software [34] that builds a decision tree, and each node selects a random number of attributes. This algorithm does not use pruning. In order to determine the best number of trees to generate the ensemble, a preliminary study of the impact in the optimization process of the size of the ensemble was performed.

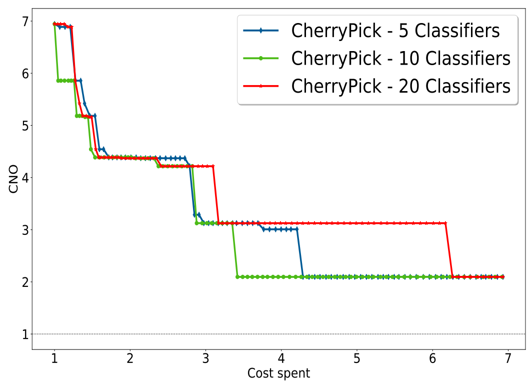
Figure 6.1 shows the CNO as a function of the cost of the optimization process using 5, 10 or 20 trees in the ensemble for training a CNN. Using Nephele, there is not a significant difference between the different experiments to train a CNN. However, when using 20 classifiers, Lynceus (with a lookahead horizon of zero) and CherryPick have an exploration cost approximately 4 times higher when compared with using 5 or 10 classifiers. To train a RNN (Figure 6.2) using 20 classifiers always increases the exploration cost, as in CNN training. Training with 5 classifiers presents better results in two out of three cases. To train a Multilayer NN (Figure 6.3), the best results are always obtained using an ensemble of 5 classifiers. Therefore, it was implemented a random forest using an ensemble containing 5 decision trees. Nephele builds two ensembles: one for modeling the cost function and the other for the accuracy function.



(a) Nephele

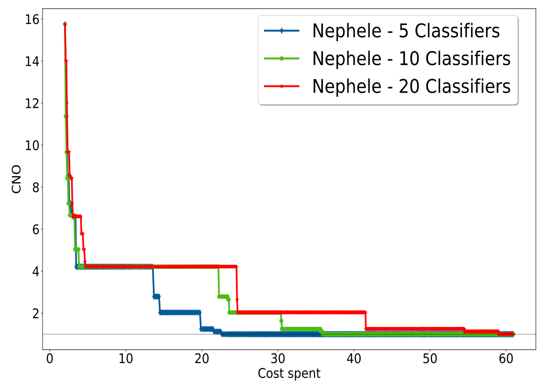


(b) Lynceus

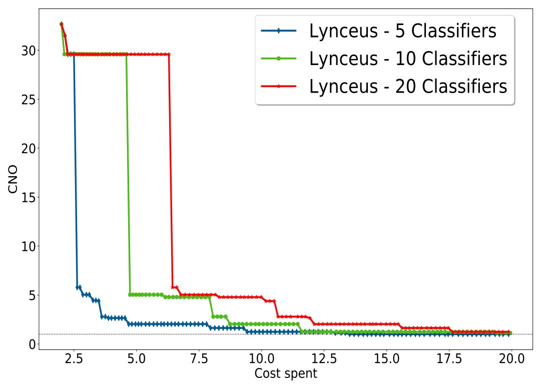


(c) CherryPick

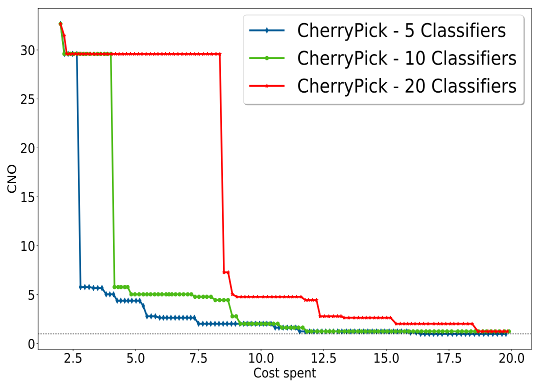
Figure 6.2: Comparison between different number of classifiers in the ensemble to train a RNN



(a) Nephele



(b) Lynceus



(c) CherryPick

Figure 6.3: Comparison between different number of classifiers in the ensemble to train a Multilayer NN

Experimental Setup. In order to construct the models that represent the initial belief about the cost and the accuracy functions, the LHS technique [78] was used. Since the search spaces and the objective functions are different when using subsampling ($\forall s \in \mathcal{D}$) or not ($s = 1$), to ensure a fairer comparison all the baselines are given the same initial budget for the initial sampling. It is crucial to choose a good value for the initial budget since there is a trade-off between spending too much budget (or too few) and building a model that is accurate enough (or too poor). Therefore, a preliminary study was conducted to evaluate the influence of the initial budget in the model predictions. All the experiments were executed in machines with the Intel Xeon Silver 4116 Processor and 64GB of memory running the operating system Linux Ubuntu 18.04 LTS with an architecture x84_64.

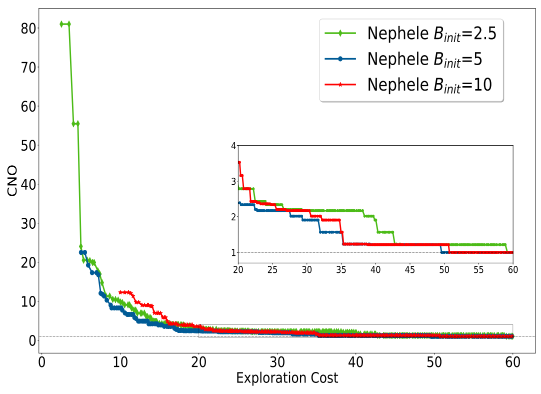
Nephele was compared against Lynceus and CherryPick and, in order to do so, three different initial budgets for each network and optimizer were tested. CNN is more expensive to train, so the initial budgets tested were higher compared with the two other networks. For each network and initial budget, the percentage of the dataset explored through the initial sampling when subsampling is used corresponds to approximately half of the percentage explored when subsampling is not used. The comparison of different initial budgets for each network is represented in the following Figures. Each job was run one hundred times with different seeds in order to mitigate the impact of the randomness of the initial sampling process in the models.

The initial budgets for training a CNN were \$2.5, \$5, and \$10, which corresponds on average to 0.76%, 1.39%, and 2.71% of the search space using the subsampled dataset, respectively. Figure 6.4 shows the 90^{th} percentile of one hundred runs of the CNO of the incumbent configuration in each iteration in function of the 90^{th} percentile of the exploration cost spent to find the optimal configuration to train a CNN. For this particular network, the initial budgets tested do not have a significant influence on the final results since the initial model does not seem to be too uncertain because it spends approximately the same amount to find the optimum.

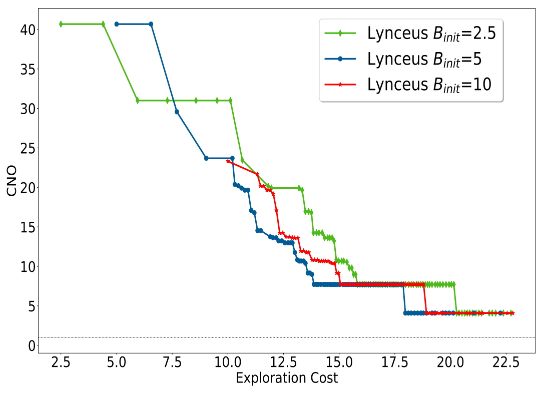
In spite of the results show that the exploration costs to find the optimal configuration are similar, through the analysis of Figure 6.4, it is possible to carefully choose an initial budget that minimizes the exploration cost in most of the cases. With $B_{init} = 2.5$, Nephele spends more money to find the optimum. There is not a significant difference between \$5 and \$10. Using Nephele and Lynceus, there is a small gain when $B_{init} = 5$ comparing with the other budgets. However, CherryPick has a better performance when $B_{init} = 2.5$. Selecting $B_{init} = 5$, the exploration cost is slightly smaller in two out of three cases and, thus, the initial budget was set to \$5.

To train a RNN, the budgets given to the initial sampling were \$0.25, \$0.5, and \$1, and Nephele evaluates, on average, 2.29%, 4.31%, and 8.13% of search space, respectively. The results obtained are represented in Figure 6.5. Using Nephele, the best result was obtained for $B_{init} = 0.25$. However, for the other two optimizers, the fastest convergence, i.e., the lower exploration cost occurs when $B_{init} = 1$. The differences between exploration cost for different initial budgets are more highlighted when no subsampling is used (i.e., with Lynceus and CherryPick). The cost spent to find the optimum for $s = 1$ can be approximately 2 and 3.5 times higher for $B_{init} = 0.25$ and $B_{init} = 0.1$, respectively, while using subsampling these differences are, approximately, 1.5 times higher.

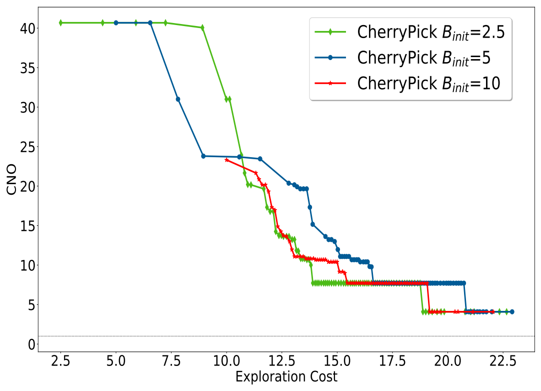
The initial budget to train a Multilayer NN was set to \$0.5, \$1, and \$2, and Nephele evaluates on average 1.94%, 3.61%, and 6.94% of the search space, respectively, for the initial sampling. The results are presented in Figure 6.6. The results obtained using an initial budget of \$0.5 and \$1 are worse comparing with $B_{init} = 2$. The initial samples do not represent the space correctly and, therefore, the model created is poor and uncertain. The predictions done based on the model about the optimum are more incorrect, and the systems need more money to improve the model and to find the optimum.



(a) Nephele

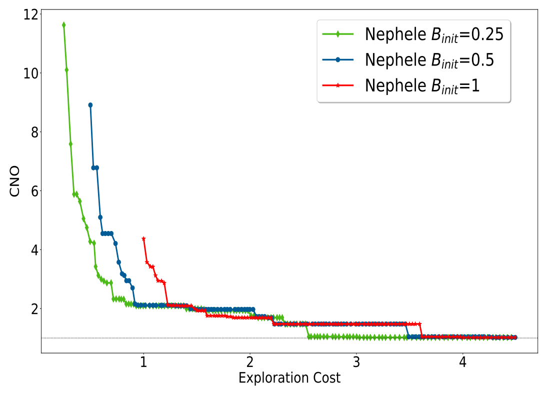


(b) Lynceus

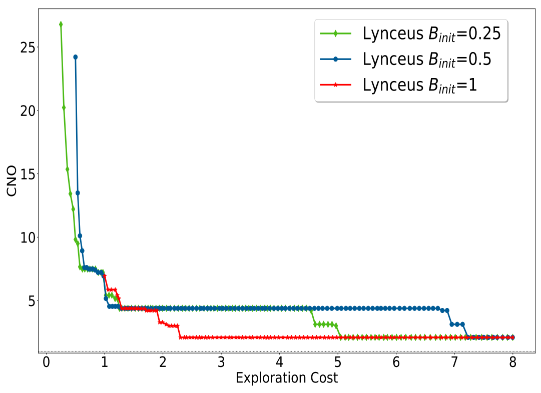


(c) CherryPick

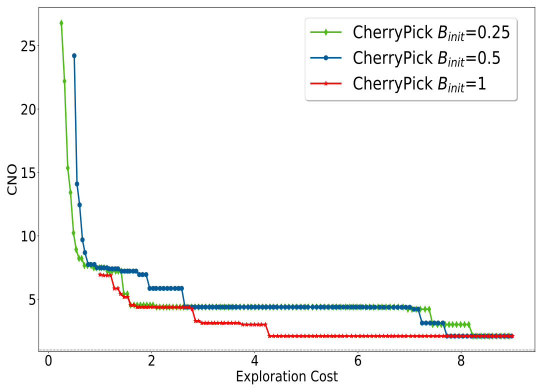
Figure 6.4: Comparison between different initial budgets to find the optimum to train a CNN



(a) Nephele

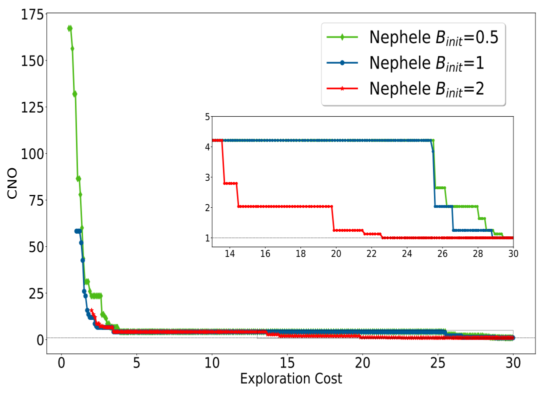


(b) Lynceus

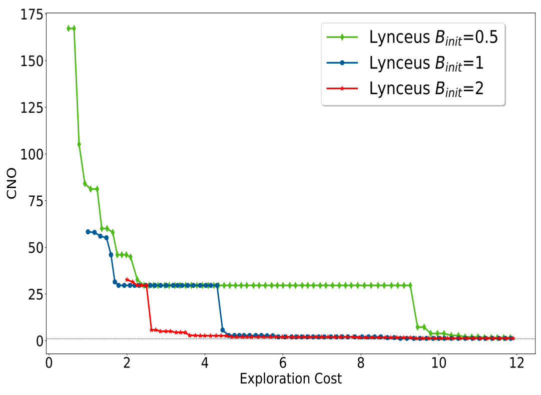


(c) CherryPick

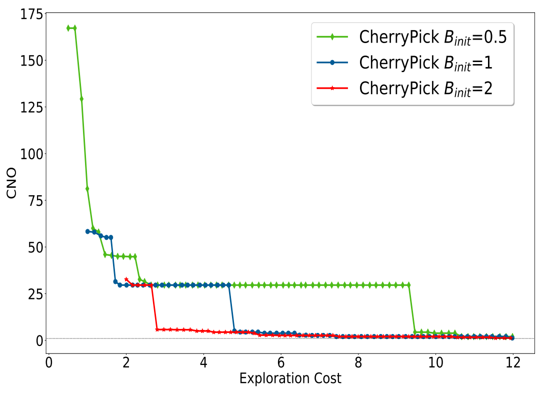
Figure 6.5: Comparison between different initial budgets to find the optimum to train a RNN



(a) Nephele



(b) Lynceus



(c) CherryPick

Figure 6.6: Comparison between different initial budgets to find the optimum to train a Multilayer NN

When Nephele uses an initial budget of \$0.5 or \$1, it spends approximately 1.3 times more compared with an initial budget of \$2 to find near-optimal and optimal configurations. For Lynceus and CherryPick, the two smaller budgets lead to 2.5 times larger expenses during exploration. Therefore, the initial budget chosen when the job deployed is the training of a Multilayer NN is \$2 because the constructed model is less uncertain and has a better representation for the objective function.

Table 6.1 summarizes the previous study on the impact of the initial budget that should be given to the initial sampling process to construct the models.

In order to evaluate the overall system's performance, the budget B was set to a value that could allow the optimizers to find the optimum. In this work, the accuracy constraint was tuned for $A_{\min} = 85\%$, as in Casimiro et al. [19]. Posteriorly, in order to choose a fair time constraint for the job deployed, the configurations (x, s) that respect the accuracy constraint were ranked by time. The results are registered in Figure 6.7, where it is possible to see that the time taken to execute the job varies from a few seconds to more than 10 minutes (which was the time threshold to run a configuration in the cloud). Based on Figure 6.7, the time constraint was set to 300 seconds (5 minutes) in order not to make it too challenging for the optimizers to find feasible configurations. This threshold allowed for 14.44%, 7.64% and 18.33% of feasible configuration in the CNN, Multilayer and RNN networks, which is already more challenging by 1.47 times in the best case and 2,64 times in the worst case than in state-of-the-art systems [19, 2].

Parameter γ (Algorithm 3, Lines 27-28) was set to 0.9 as proposed in Lam et al. [53]. The maximum lookahead horizon was always set to zero either with Lynceus or Nephele in the following results, except in Section 6.5, which is focused precisely on evaluating the impact of lookahead.

Network	Initial Budget	s	No. of initially sampled configurations (on average)	% of search space sampled
CNN	\$5	$\forall s \in \mathcal{D}$	20	1.4
		$s = 1$	8	2.8
RNN	\$1	$\forall s \in \mathcal{D}$	117	8.13
		$s = 1$	44	15.3
Multilayer	\$2	$\forall s \in \mathcal{D}$	100	6.9
		$s = 1$	32	11.1

Table 6.1: Percentage of search space sampled with the initial budgets for each network

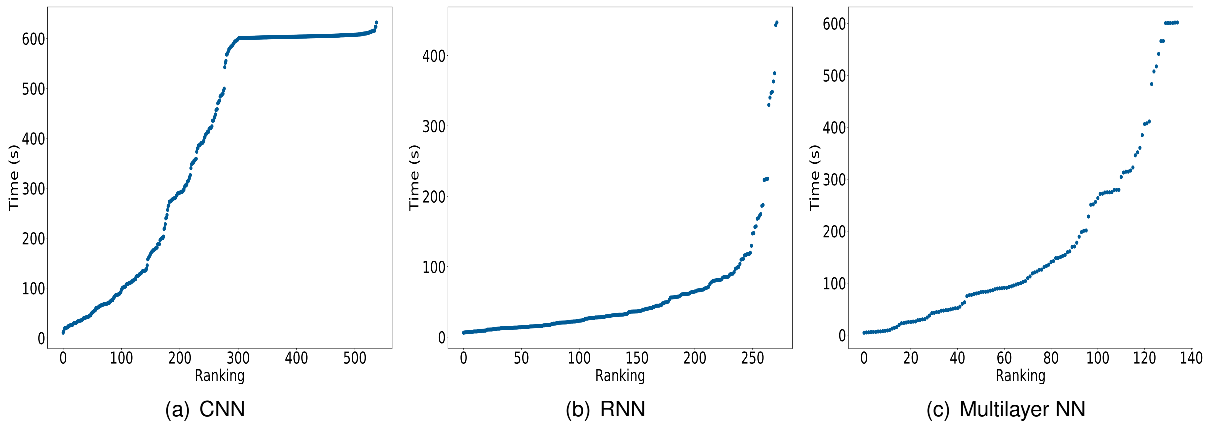


Figure 6.7: Configuration meeting the accuracy constraint sorted by execution time.

6.3 Impact of Subsampling on Cost in Production

This section addresses the first question raised in Chapter 1: can the use of subsampling reduce the cost in production, i.e., is it possible to identify configurations that satisfy the quality constraints and achieve a cost reduction? This section studies the effect on the deployment cost of using subsampling techniques in order to answer this question. For that, we start by analyzing the search space for each network.

Typically, by using the full dataset to train a neural network, it is possible to obtain better accuracy values because there is more data available. However, this may be more expensive than training smaller datasets because there is more data available that need to be observed in the training process. However, if the accuracy constraint is too high, it is more challenging to ensure a good enough performance by using smaller datasets. The accuracy and time constraints are very dependent on the application that is being deployed. A job can need several minutes or hours to be run, e.g., training neural networks, or can take days or months, e.g., simulation of biological systems. Also, depending on the performance required by the user, a job can be less (and usually cheaper) or more accurate.

Let $\mathcal{D}$ be a set that contains the possible values of s. When subsampling is not used, i.e., the full dataset is used to train, the value of s is 1. When subsampling is used, s may have a value contained in $\mathcal{D}$. Therefore, the notation used in this work when subsampling is used is $\forall s \in \mathcal{D}$. Figure 6.8 presents a heat map, which allows for a better analysis of the possible cost improvements brought by the use of subsampling for different values of the constraints. For each possible combination of the constraints (A_{mim}, T_{max}), it is calculated the cost reduction of running a optimal configuration x^* using subsampling

Network Type	No. of Feasible Configurations		% of feasible search space	
	$\forall s \in \mathcal{D}$	$s = 1$	$\forall s \in \mathcal{D}$	$s = 1$
CNN	208	61	14.44%	21.18%
RNN	264	104	18.33%	36.1%
Multilayer	110	58	7,64%	20.14%

Table 6.2: Feasible Configurations

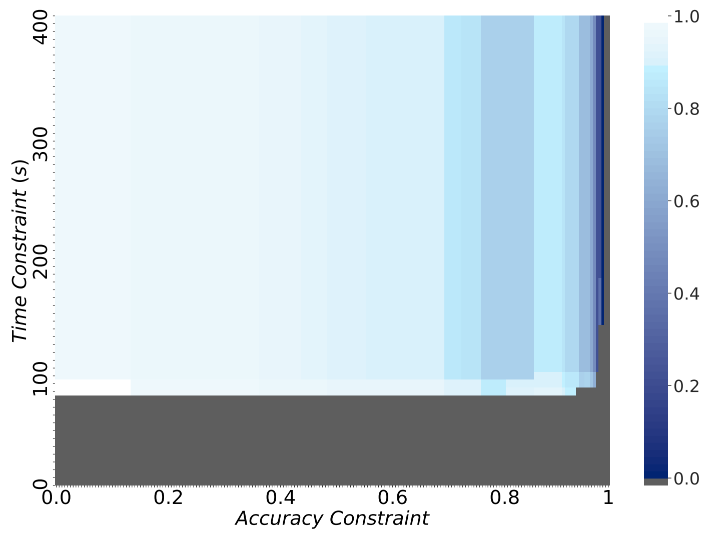
($\forall s \in \mathcal{D}$) or not ($s = 1$). The cost reduction is given by the difference between the cost of the optimum using the full dataset ($C(x^*, s = 1)$) and the optimal cost using all possible sizes of subsampled datasets ($C(x^*, \forall s \in \mathcal{D})$) normalized with respect to the optimum on the full dataset i.e.,

$$\frac{C(x^*, s = 1) - C(x^*, \forall s \in \mathcal{D})}{C(x^*, s = 1)}. \tag{6.2}$$

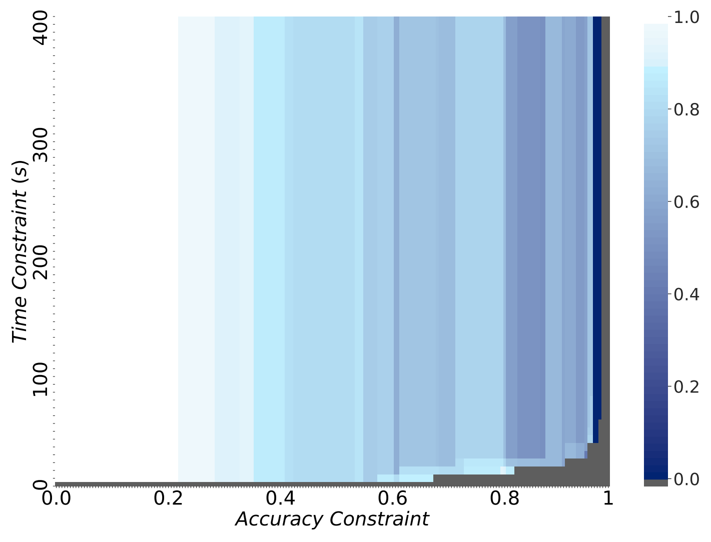
When it is not possible to use subsampled datasets to solve the optimization problem, Equation 6.2 is equal to 0 because it is not possible to achieve any cost reduction through subsampling. When the optimal configuration is in subsampled datasets, the normalized cost reduction is a value between 0 and 1. The space colored in grey in Figure 6.8 represents pairs of accuracy and time constraints that create an infeasible optimization problem. When the normalized cost reduction achieved by subsampling is high and closer to one, it is much cheaper to train the network with subsampled datasets, and this happens, as expected, when the accuracy constraint is small, i.e., the higher cost reductions can be achieved for lower accuracy thresholds. Therefore, by relaxing the accuracy requirement, the optimal cost decreases, making it possible to use subsampled and cheaper datasets that respect the QoS constraints. When high accuracy is required, the cost increases, and it may be necessary to train with the full dataset to ensure the QoS. If the accuracy constraint is too high, it may be impossible to ensure the QoS constraints. For example, when the defined accuracy constraint to train a Multilayer NN is higher than, approximately, 90%, the optimization problem is infeasible (region colored in grey in Figure 6.8(c)). On the contrary, when the time constraint is too small, it may not be possible to solve the optimization problem. This problem is emphasized on CNN because this network takes more time to train than the other two. Analyzing Figure 6.8(a), when the time constraint is lower than approximately 60 seconds, the problem becomes infeasible because the network does not have enough time to be trained. The same happens when training a Multilayer and a RNN, but the values of time constraint that create infeasible problems are much smaller.

In the following, the accuracy constraint was set to 85% and the time constraint to 5 minutes. Therefore, the majority of configurations do not respect the constraints. The percentage of feasible configurations is calculated in Table 6.2 with and without subsampling. Using subsampling datasets, the smaller the feasible region is, the harder it is to find the optimum.

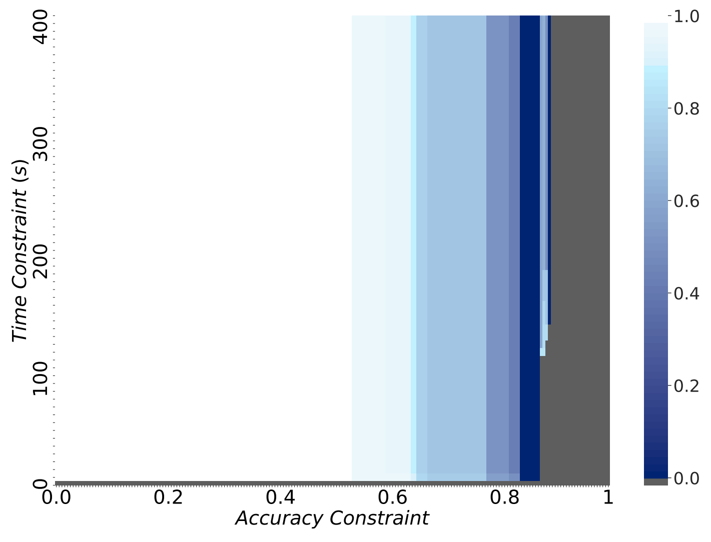
Next, in Figure 6.9 each network is analyzed separately for different dataset sizes When the dataset size increases, there is a decrease in the number of configurations that respect the time constraint. For the same configuration, increasing the dataset size implies that the number of iterations required to train the network also increases. This provokes an increase in the training time and, therefore, leads to an increase in the number of configurations that do not respect the time constraint. On the other hand, the reduction of the dataset size causes a decrease in the training time of the network because it requires fewer iterations and, therefore, fewer samples are used to train. These two reasons may cause a decrease in the accuracy of a given configuration. If the dataset is too small, there may be no feasible configurations. This happens when the RNN and the Multilayer networks are trained using only 1.6% of



(a) CNN

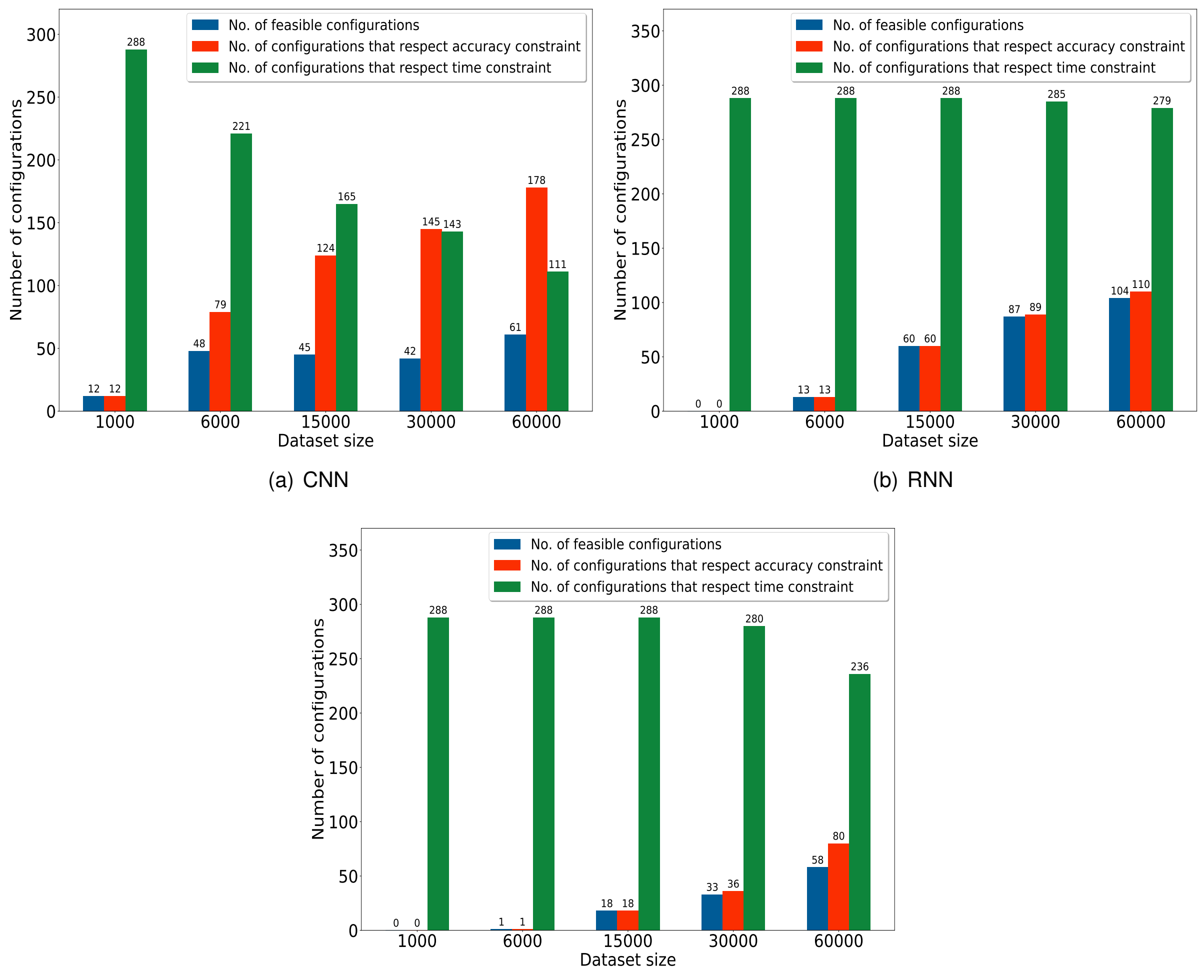


(b) RNN



(c) Multilayer NN

Figure 6.8: Cost reduction achieved using subsampling normalized w.r.t. the optimal cost on the full dataset



(a) CNN

(b) RNN

(c) Multilayer NN

Figure 6.9: Number of feasible configurations for different dataset sizes

the full dataset. In this case, the datasets are too small and do not have enough information to train the network correctly.

6.3.1 CNN Dataset

Training a CNN using subsampled datasets, the number of configurations that respect the accuracy constraint increases with dataset size, and the opposite effect happens with the time constraint. However, the increase of the dataset size does not necessarily imply an increase in the feasible space, as it is possible to see in Figure 6.9(a).

The use of subsampling to train a CNN allows for reductions of 98.33% in the dataset size, while still allowing both constraints to be complied with. Using the previously defined constraints and subsampled datasets, only 14.44% of the configurations are feasible for this network. Table 6.3 compares the feasible configurations, configurations that respect one of the constraints, and infeasible configurations for the different subsampled datasets with the full dataset. $Time/Acc$ means that both time and accuracy constraints are satisfied for a given configuration in subsampled datasets. $Time/\overline{Acc}$ and $\overline{Time}/Acc$ means that only time or accuracy constraint are respected, respectively. $\overline{Time}/\overline{Acc}$ means that both constraints are violated. Decreasing the dataset size leads to an increase in the number of configurations

1K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	0 (0)	0	21.18% (61)	0
$\overline{Time}/Acc$	4.17 % (12)	0	36.46% (105)	0
$Time/\overline{Acc}$	0	0	17.36% (50)	0
$\overline{Time}/\overline{Acc}$	0	0	20.83% (60)	0

(a) 1000 vs. 60000

6K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	1.74% (5)	0	19.44% (56)	0
$\overline{Time}/Acc$	14.94% (43)	9.72% (28)	13.88% (40)	2.08% (6)
$Time/\overline{Acc}$	0	0	17.36% (50)	0
$\overline{Time}/\overline{Acc}$	0	1.04% (3)	9.38% (27)	10.42% (30)

(b) 6000 vs. 60000

15K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	6.94% (20)	0	14.24% (41)	0
$\overline{Time}/Acc$	8.69% (25)	25.35% (73)	4.51% (13)	2.08% (6)
$Time/\overline{Acc}$	0	0	17.36% (50)	0
$\overline{Time}/\overline{Acc}$	0	2.08% (6)	5.56% (16)	13.19% (38)

(c) 15000 vs. 60000

30K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	12.15% (35)	0	9.03% (26)	0
$\overline{Time}/Acc$	2.43% (7)	33.68% (97)	3.13% (9)	1.39% (4)
$Time/\overline{Acc}$	0	0	17.36% (50)	0
$\overline{Time}/\overline{Acc}$	0	2.08% (6)	5.56% (16)	13.19% (38)

(d) 30000 vs. 60000

Table 6.3: Comparison between different subsampled dataset and the full dataset using CNN. Each cell has the number of configurations and the respective percentage of configurations of the search space that respect a given combination of the time and accuracy constraints in the subsampled and the full dataset.

that satisfy the time constraint. However, fewer configurations meet the accuracy constraint. The number of configurations that respect the same constraints in the full dataset and the subsampled datasets decreases with the reduction of s (this can be observed by analyzing the diagonal of each subtable of Table 6.3 and noticing that the number of configurations diminishes). Overall, using the smallest considered dataset with 1000 images, all configurations meet the time constraints. Additionally, 95.83% of the configurations also do no comply with the accuracy constraint for such a small subsampled dataset ($s = 0.0167$). Using 10% of the full dataset, the number of feasible configurations grows. However, there are more configurations that do not respect the time constraint.

The dataset size specified in the optimal configuration that minimizes the deployment cost to train a CNN and ensures the QoS constrains is $s = 0.1$ (dataset with 6000 images). The additional cost to pay is 4.1 times higher when subsampling is not used. Therefore, for training a CNN, it is possible to use subsampled datasets and reduce the deployment cost of the optimal configuration in 75.5%.

6.3.2 RNN Dataset

The training of a RNN, the number of configurations that respect the accuracy constraint increases exponentially with the dataset size, and the number of configurations that meet the time constraint decreases only for larger datasets. The feasible space also increases with s, and only 18.33% of the search space is feasible.

Table 6.4 compares the performance of training a RNN for the different values of s. More than 99.1% of the configurations respect the time constraint, and the violations occur when the network is trained using the whole and half of the dataset. However, when training with 1.6% of the full dataset, the accuracy constraint is always violated. Approximately 60% of the configurations respect the time constraint in both the subsampled and full dataset. However, these configurations violate the accuracy constraint. The number of configurations that comply with QoS constraints in the subsampled and the full datasets increases with the size.In this network, it is more difficult to ensure good enough accuracy with small datasets. If the dataset used to train is too small ($s = 0.016$), it is impossible to ensure the QoS. However, there are configurations that respect both constraints and that were trained with subsampled datasets. The optimal configuration that minimizes deployment cost and respects the QoS constraints can be found in the dataset containing 15000 images ($s = 0.25$). Thus, it is possible to use subsampled datasets to achieve cost reduction. When subsampling is not used, the additional cost to pay is approximately 2.1 times higher. For this particular dataset and optimization problem, it is possible to reduce the cost in production by 52.3%.

6.3.3 Multilayer NN Dataset

When the job run is the train of a Multilayer NN, there is an exponential increase in the number of feasible configurations with s. The number of configurations that respect the accuracy constraints increases with the dataset size. However, since this network needs less time to be trained, which can be confirmed in Figure 6.9(c), for smaller values of s, the time constraint does not affect the number of feasible configurations.

Table 6.5 analyzes each subsampled dataset corresponding to the training of a Multilayer NN, as previously done. All configurations respect the time constraint when subsampled datasets are used. Using 30000 images for training, less than 3% of the configurations violate the time constraint, and

1K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	0	0	36.11% (104)	0
$\overline{Time}/Acc$	0	0	2.08% (6)	0
$Time/\overline{Acc}$	0	0	60.77% (175)	0
$\overline{Time}/\overline{Acc}$	0	0	1.04% (3)	0

(a) 1000 vs. 60000

6K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	3.47% (10)	0	32.64% (94)	0
$\overline{Time}/Acc$	1.04% (3)	0	1.04% (3)	0
$Time/\overline{Acc}$	0	0	60.77% (175)	0
$\overline{Time}/\overline{Acc}$	0	0	1.04% (3)	0

(b) 6000 vs. 60000

15K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	19.09% (55)	0	17.01% (49)	0
$\overline{Time}/Acc$	1.74% (5)	0	0.35% (1)	0
$Time/\overline{Acc}$	0	0	60.77% (175)	0
$\overline{Time}/\overline{Acc}$	0	0	1.04% (3)	0

(c) 15000 vs. 60000

30K 60K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	28.82% (83)	0	7.29% (21)	0
$\overline{Time}/Acc$	1.39% (4)	0.69% (2)	0	0
$Time/\overline{Acc}$	0	0	60.77% (175)	0
$\overline{Time}/\overline{Acc}$	0	0	0.69% (2)	0.35% (1)

(d) 30000 vs. 60000

Table 6.4: Comparison between different subsampled dataset and the full dataset using RNN

training with the entire dataset this value increases to 18%. The number of configurations that meet the time constraint but violate the accuracy constraint does not vary with s. With the increase of the dataset size, there are some configurations that respect both constraints in the full and subsampled datasets. However, there is a large number of configurations (61.80%) that do not meet the minimum accuracy required in any dataset. When a set of 1000 images is used to train, all configurations do not reach the desired accuracy value because the dataset does not have enough information to ensure the accuracy constraint. Increasing the size of the dataset, the number of feasible configurations in both dataset sizes increases slightly.

When subsampling is used, the feasible region for this network is small (7.64% of configurations are feasible), which means that it is not possible to reduce the dataset to the same size as CNN and, at the same time, accomplish the constraints. For example, when the dataset is reduced to 1000 images ($s = 0.0167$), it is not possible to ensure the QoS and when the dataset has 6000 images ($s = 0.1$), there is only one configuration that respects the constraints. Although, there are feasible configurations for s equal to or greater than 0.1, the optimal configuration that minimizes the cost subject to the QoS constraints is in the full dataset ($s = 1$). Therefore, in this case, it is not possible to decrease the cost of production using subsampling for the given optimization problem.

6.4 Impact of Subsampling on Cost of Optimization Process

This section analyzes the impact of using subsampling in the exploration cost, i.e., cost spent to evaluate different configurations to find the optimum, in order to answer to the question raised: can the use of subsampling techniques reduce the cost of the optimization process to find a feasible solution at a distance d from optimum?

As concluded in Section 6.3, it is possible to use subsampled datasets, which have less information than the full dataset but are cheaper to evaluate, in order to reduce the deployment cost of running a job in the cloud while ensuring the user-specified QoS constraints. However, the search space created using small datasets is five times bigger than using only the full dataset. There is one more dimension in the search space and, instead of 288 possible configurations, there are 1440 configurations. Therefore, since the input space is larger and more complex, the exploration phase might take more time or require more explorations and, thus, there is a probability to be more expensive.

Nephele, Lynceus, CherryPick, and Random Search were used to solve the optimization problem for the three datasets in order to study the impact of subsampling in the exploration cost, time, and number of explorations.

6.4.1 CNN Dataset

Figure 6.10 shows the CNO as a function of cost, time and number of explorations of the training of a CNN using the constraints defined in Section 6.2. Through the analysis of Figures 6.10(a) and 6.10(b), it is possible to see that for the same exploration cost or time Nephele can find better configurations with lower cost that meet the QoS, spending less money and taking less time. For example, to achieve a CNO equal to ten, Lynceus and CherryPick spend 1.75 and 2.13 times more than Nephele. The initial budget for the initial sampling to construct the models is the same for all the optimizers. Although, Nephele explores more configurations comparing with Lynceus and CherryPick, the configurations explored by

60K \ 1K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	0	0	20.14% (58)	0
$\overline{Time}/Acc$	0	0	7.64% (22)	0
$Time/\overline{Acc}$	0	0	61.80% (178)	0
$\overline{Time}/\overline{Acc}$	0	0	10.42% (30)	0

(a) 1000 vs. 60000

60K \ 6K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	0	0	20.14% (58)	0
$\overline{Time}/Acc$	0.35% (1)	0	7.29% (21)	0
$Time/\overline{Acc}$	0	0	61.80% (178)	0
$\overline{Time}/\overline{Acc}$	0	0	10.42% (30)	0

(b) 6000 vs. 60000

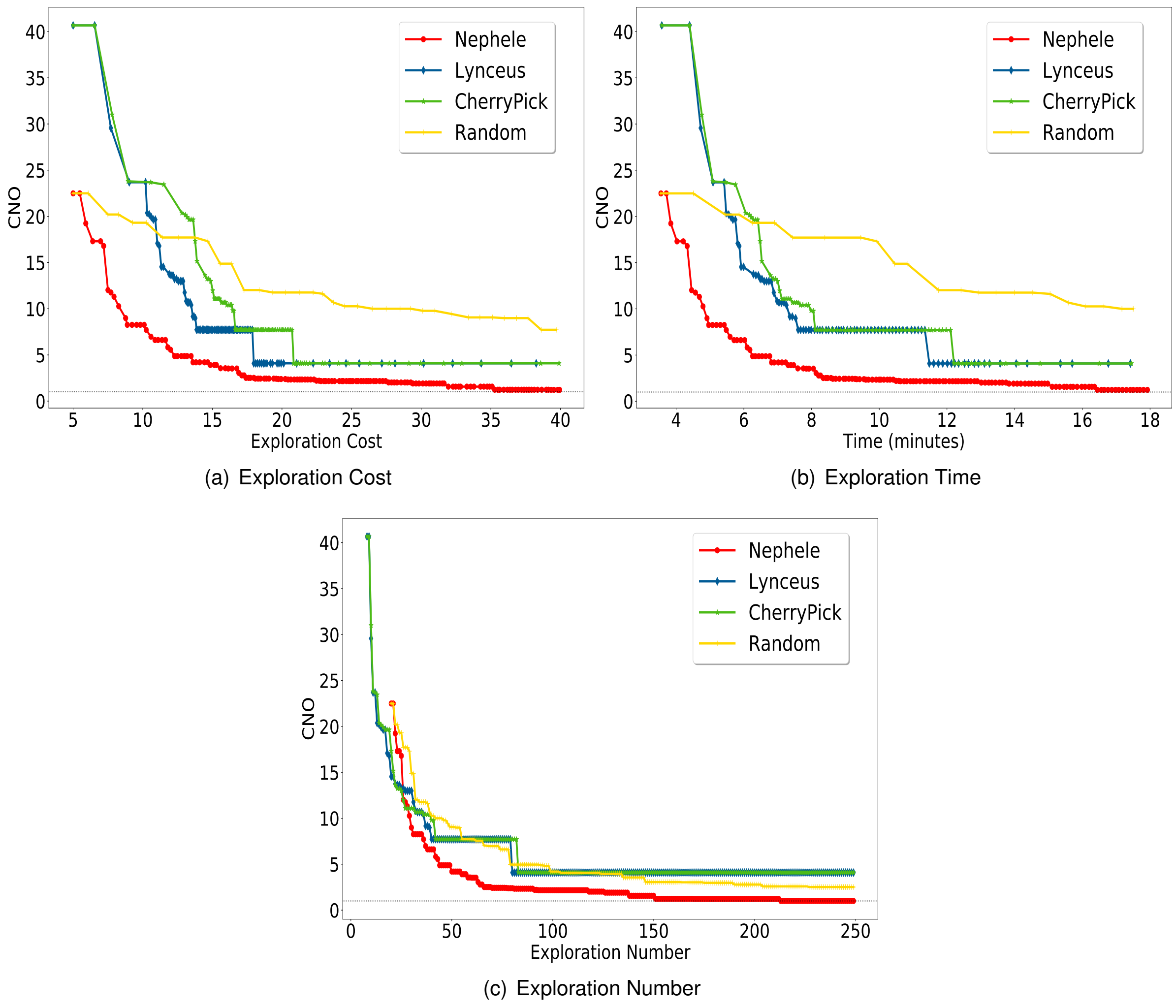
60K \ 15K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	3.82% (11)	0	16.32% (47)	0)
$Time/\overline{Acc}$	2.43% (7)	0	5.21% (15)	0
$Time/\overline{Acc}$	0	0	61.80% (178)	0
$\overline{Time}/\overline{Acc}$	0	0	10.42% (30)	0

(c) 15000 vs. 60000

60K \ 30K	$Time/Acc$	$\overline{Time}/Acc$	$Time/\overline{Acc}$	$\overline{Time}/\overline{Acc}$
$Time/Acc$	9.03% (26)	0	11.11% (32)	0
$\overline{Time}/Acc$	2.08% (6)	1.04% (3)	4.17% (12)	0.35% (1)
$Time/\overline{Acc}$	0	0	61.80% (178)	0
$\overline{Time}/\overline{Acc}$	0.35% (1)	0	8.68% (25)	1.39% (4)

(d) 30000 vs. 60000

Table 6.5: Comparison between different subsampled dataset and the full dataset using Multilayer NN



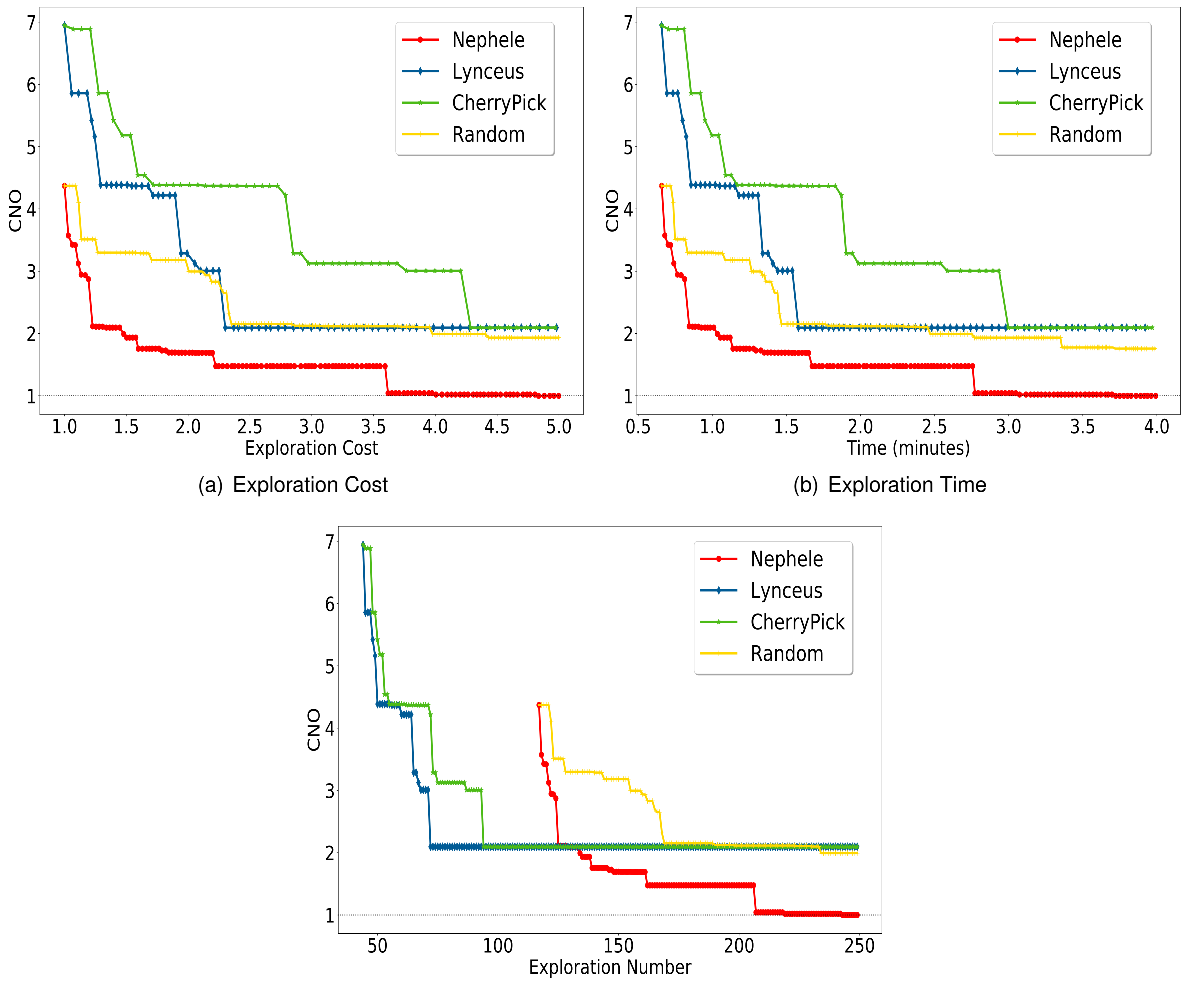
(a) Exploration Cost (b) Exploration Time (c) Exploration Number

Figure 6.10: Optimization Process to train a CNN

Nephele can make use of subsampling and are, therefore, cheaper to test on average. There, at the of the initial sampling phase, the CNO of Nephele is approximately half of the value achieved by Lynceus and CherryPick when they end their respective initial sampling phases and begin to use the performance models to guide the exploration. Using the random selection of configurations to sample, the cost of the incumbent is around ten times higher compared with when Nephele finds the optimum. Therefore, through the analysis of the obtained results, it is possible to conclude that the cost of the optimization process for training a CNN can be reduced through the use of subsampling. With the same budget, Nephele always recommends better configurations than Lynceus and CherryPick.

6.4.2 RNN Dataset

In Figure 6.11, it is possible to observe the CNO of the incumbent configuration predicted in each iteration over the cost, time, and exploration number of the optimization process. CherryPick is always the worst to find the optimum in the respective search space. During the entire optimization process, Nephele is able to recommend better configurations than the other optimizers. These configurations have lower costs, i.e., are closer to the optimum and ensure that the QoS constraints are complied with. Lynceus spends 1.2 times more than Nephele, and CherryPick spends 3.67 times more to recommend



(a) Exploration Cost

(b) Exploration Time

(c) Exploration Number

Figure 6.11: Optimization Process to train a RNN

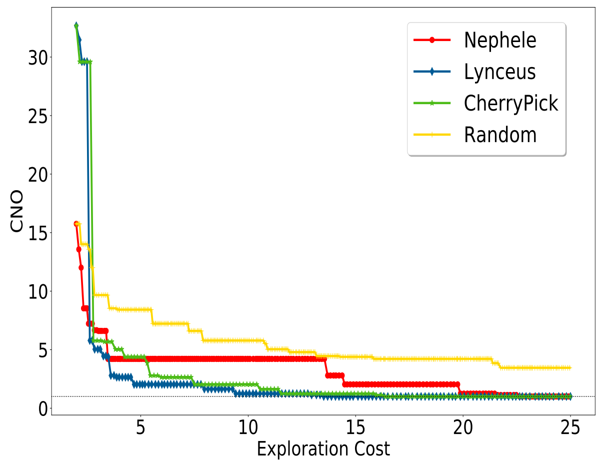
a configuration with a CNO of 2. Using the same initial budget, Nephele can explore 2.66 more configurations than Lynceus and CherryPick. Hence, the use of subsampling can reduce the cost of the optimization process for training a RNN.

Using the same amount for exploring configuration in the search space, Nephele can always recommend cheaper configurations. In the best case, Nephele can recommend configurations that have a deployment cost 3 and 3.5 times lower comparing with Lynceus and CherryPick, respectively. When Nephele finds the optimal configuration, it can achieve a reduction of 50% in the deployment cost (see Section 6.3).

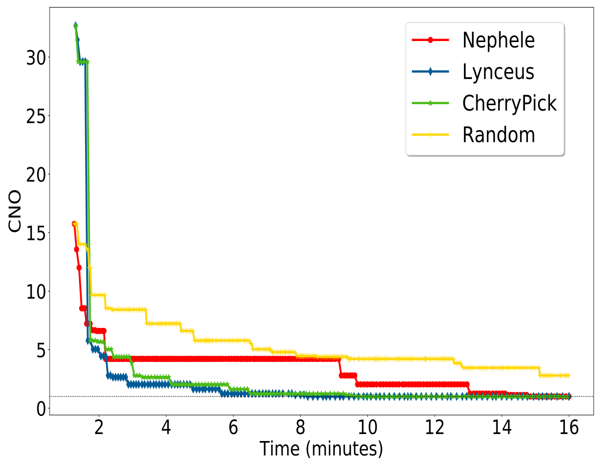
6.4.3 Multilayer NN Dataset

Figure 6.12 depicts the evolution of the optimization process over cost, time, and exploration number. As seen in Section 6.3.3, for the defined QoS constraints, the optimum is found using the full dataset, so it is not possible to use subsampling to decrease the cost in production. Since the search space increases five times, and it is much complex, Nephele consumes more money and takes longer to find the optimum.

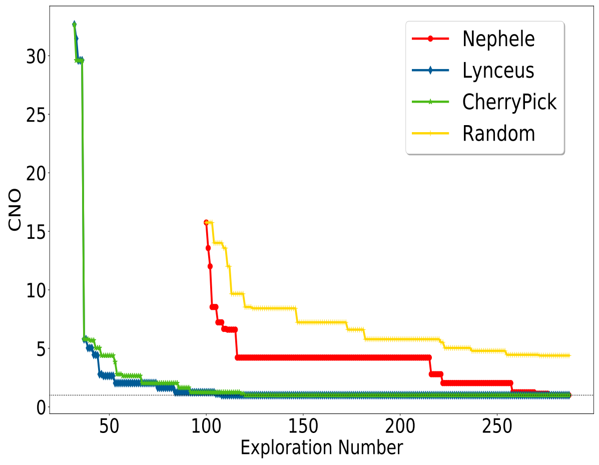
This can be explained by considering that the search space increases in size and complexity, and



(a) Exploration Cost



(b) Exploration Time



(c) Exploration Number

Figure 6.12: Optimization Process to train a Multilayer NN

the optimizer will require more time and money to find the optimum. However, in the early phases of the optimization process, where the spent budget is small (e.g., \$2.5), Nephele can recommend better configurations than the other two systems.

6.5 Impact of Subsampling on Lookahead

This section evaluates the impact of using subsampling with lookahead techniques in order to see if lookahead can have benefits when using subsampling. Since the use of subsampling creates a larger and more complex search space, which have been previously indicated as favourable conditions for the use of lookahead techniques [[53], [19]], with this study we intend to verify whether this is actually the case also for our datasets. Three depth horizons were tested: $h = 0, 1, 2$. To compute the Gauss-Hermite quadrature, it is used three points for a cost predictions and one point for the accuracy (i.e., the mean value) predictions ($N = 3$ and $M = 1$, Algorithm 3 Lines 16 and 17). A preliminary study using $M = 1$ or $M = 3$ showed that using $M = 1$ Nephele can decrease the optimization cost. Also, this way, the complexity decreases exponentially, and the optimization time is much shorter.

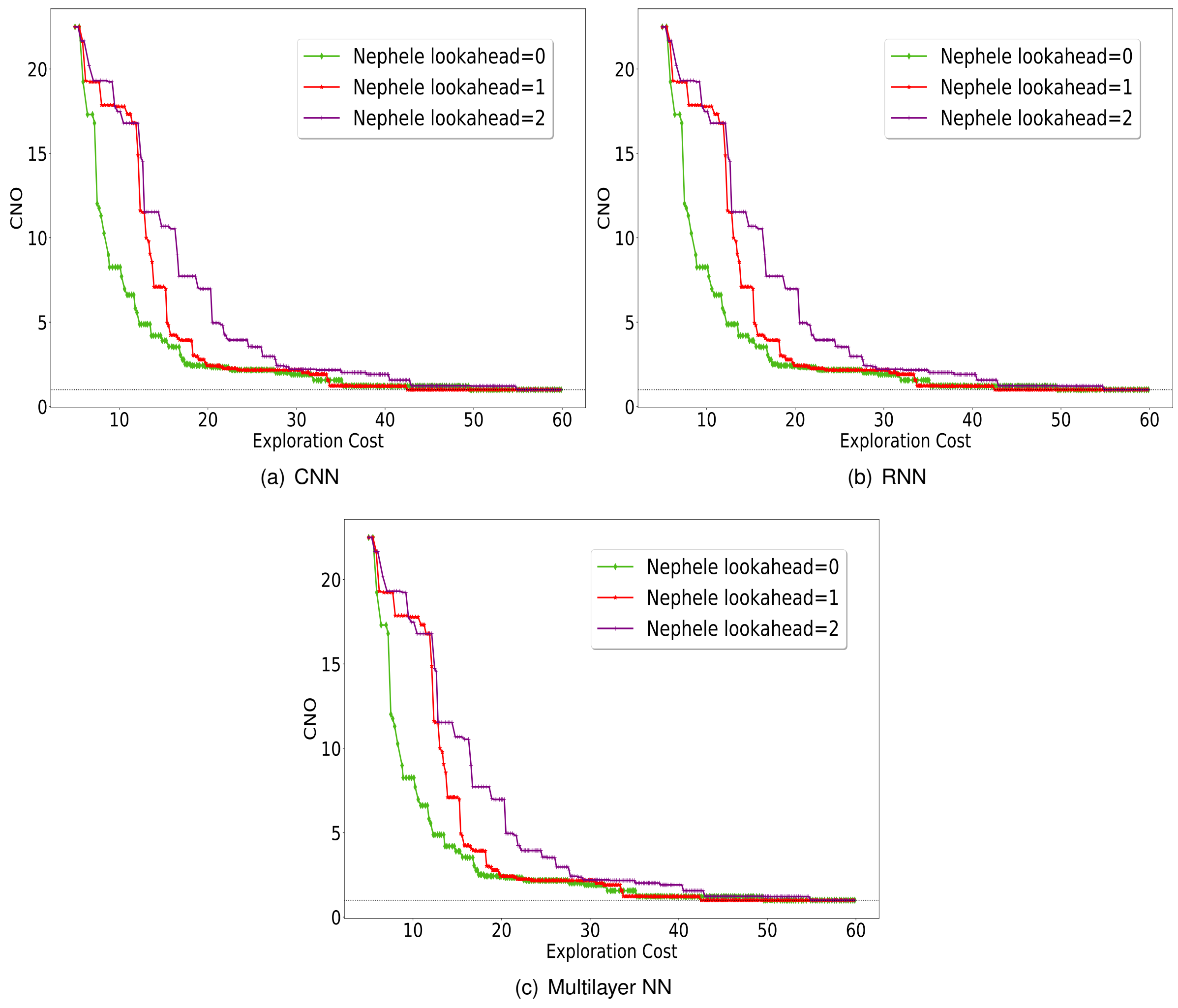


Figure 6.13: Optimization process using Nephele and different depth horizons to train NN

6.5.1 CNN Dataset

Figure 6.13(a) shows the optimization process to train a CNN using three different depths horizons. Using a horizon of two is always the worst, and the recommended configurations have a higher CNO. With a horizon of one, Nephele can spend less to find the optimal configuration. However, this cost gain is very small. It is possible to achieve better recommendations for lower exploration costs if Nephele is run without lookahead. In the best case, without using lookahead can reduce the cost in production by approximately 57%.

The total optimization time (i.e., exploration time plus the overhead necessary to select a configuration do evaluate) is 2.34 and 3.83 times higher when using a horizon of one and two comparing without lookahead.

6.5.2 RNN Dataset

Figure 6.13(b) presents the evolution of the CNO during the optimization using Nephele with three different depth horizons to train a RNN. Similar to the train of a CNN, the lookahead technique does not present significant improvements. Furthermore, using a horizon of two, the optimizer spends approximately 1.3 times more to find the optimum and during the optimization process always recommends expensive configurations.

The cost spends to find the optimum using a horizon of zero and one is similar. However, for small explorations cost, Nephele without lookahead recommends better configuration with a lower CNO.

6.5.3 Multilayer NN Dataset

In Figure 6.13(c) shows the optimization process for training a multilayer NN using different depth horizons. The cost spent by the optimizer to find the optimal configuration to train a Multilayer NN using the three possible depth horizons is very similar. However, there is a small gain when it is used a horizon of one.

During the optimization process, three is an improvement when the lookahead technique is used. Through the use of a horizon of one, it is possible to find near-optimum configurations using almost half of the cost than using no horizon. Using a horizon of 2, there is a cost reduction of 15% comparing without horizon. Therefore, in this particular case, using the lookahead technique is beneficial, and it is possible to recommend better configuration with lower CNO spending less money.

However, the total optimization time increases when lookahead is used, if one accounts also for the time necessary to predict the next configuration to test. Therefore, using a horizon of zero, the total time is almost half of the time consumed when it is used a horizon of 1. Using a horizon of 2, the total optimization time is three times higher when no horizon is used.

In general, the use of the lookahead technique does not seem to improve the performance of Nephele significantly, at least for the considered datasets. Furthermore, using and horizon of two is worst and will present a high cost for the user. In some cases, lookahead can improve the quality of recommendation during the optimization. Using the datasets analyzed in this section, in two out of three jobs, lookahead does not present significant improvements.

Summary

This chapter answered three main questions:
- can the use of subsampling reduce the cost in production, while preserving acceptable accuracy levels?
- can the use of subsampling reduce the cost and duration of the optimization process?
- is the use of lookahead techniques beneficial to improve the efficiency of the optimization process when subsampling is used?

The results of our study show that the answer to the first two question is yes, provided that the accuracy constraints can be actually satisfied using subsampled datasets. This is the case for two networks out of the three considered in our study (CNN and RNN). For example, to train a CNN spending $10 to explore, Nephele can find configurations three times cheaper and, when Nephele finds the optimum, the configuration leads to a reduction of 75% of the cost in production when compared to Lynceus. To train a RNN, Nephele can always recommend better configurations and an optimal configuration approximately two times cheaper comparing to the other two optimizers. However, for training a Multilayer NN, it is not possible to satisfy the considered accuracy constraint using any subsampled datasets. Also, since the search space that includes the subsampling rate is five times larger and much more complex than the full dataset, the optimizers that do not use subsampling will have lower exploration costs.

As for the third question, using lookahead with subsampling does not present significant improvements in the quality of the recommendations and the cost spent in the optimization process, at least with the consiedred datasets. Only when training a Multilayer NN and using horizon one, the optimizer can recommend better configurations using small exploration costs. Also, with this horizon, it is possible to reduce the cost necessary to find the optimum, though this reduction is very small.

7 Fabulinus Evaluation

This chapter describes the experiments performed to test and evaluate the Fabulinus system proposed in Chapter 4. Similarly to the previous chapter, Section 7.1 provides the baselines and the metrics used for comparing Fabulinus. Section 7.2 provides the implementation details to run Fabulinus. Section 7.3 evaluates Fabulinus in order to understand the impact of subsampling to reduce the cost of the optimization process. Then, in order to validate the new acquisition function proposed, the ability of Fabulinus to find the optimal configuration is evaluated.

7.1 Evaluation Setup

This section provides the baselines to compare Fabulinus. Firstly, it defines the state-of-the-art systems with which Fabulinus is compared and, then, presents the metric used to evaluate and compare the systems.

Baselines for Comparison. Fabulinus is compared with Fabolas [48]. Fabulinus extends the acquisition function of proposed in Fabolas, called Entropy Search (Equation 2.25), in order to incorporate constraints. This new acquisition function proposed in this work is called constrained Entropy Search ($\mathrm{ES_C}$). Fabulinus is also compared to standard BO without subsampling using the constrained Expected Improvement ($\mathrm{EI_C}$) and the Expected Improvement (EI) as an acquisition function. All these systems use BO to find the optimum, and two GPs models are used to model both the cost and the accuracy functions.

Evaluation Metrics. In order to evaluate Fabulinus, a metric called constrained Accuracy $Accuracy_C$, based on similar approaches used in Lam et al. [53], is employed. At each iteration, the constrained accuracy penalizes configurations that do not meet the constraints.

$$Accuracy_C(x,s) = \begin{cases} A(x,s) & \text{if } (x,s) \text{ is feasible} \\ 0 & \text{otherwise.} \end{cases} \tag{7.1}$$

Therefore, when a configuration is infeasible, its accuracy value is penalized to zero in order to punish infeasible recommendations. The $Accuracy_C$ is evaluated as a function of the exploration cost, number of explorations, exploration time, and total computational time, which consists of the exploration time plus the overhead time necessary to compute the acquisition function to select the next configuration to evaluate.

7.2 System Implementation and Experimental Setup

This section details the system implementation to evaluate Fabulinus and the settings used to run the experiments.

Network Type	Cost constraint	Feasible configurations	Configurations with high accuracy	Configurations with high accuracy that respect constraint
CNN	\$0.1	111 (38.54%)	178 (61.8%)	69 (23.96%)
RNN	\$0.02	178 (61.8%)	110 (38.19%)	42 (14.58%)
Multilayer	\$0.06	161 (55.8%)	80 (27.78%)	25 (8.68%)

Table 7.1: Feasible configurations given a cost constraint

System Implementation. Fabulinus was developed in Python and extends the publicly available Fabolas implementation GPs are used to model the accuracy and the cost functions and are implemented using the Python library George. In order to find the best hyperparameters of GPs, the MCMC algorithm is used through the EMCEE package. The models use the logarithmic scale, as proposed in Fabolas.

Experimental Setup. The initial models of accuracy and cost are built using a number of configurations that corresponds to 3% of the search space size, randomly selected, as done in Casimiro et al. [19]. The stopping condition defined for these experiments was the maximum number of iterations. This value was set to 100 iterations. Since the computational cost of Fabulinus and Fabolas is very high (it requires a considerable amount of time to compute the acquisition functions and, thus, the total time is enormous and can take several days to solve the optimization problem), each optimizer was run ten times, as in Klein et al. [48].

The jobs deployed were the training of three NNs, as described in Chapter 5. Thus, different cost constraints were set for each NN. The cost constraints are \$0.1, \$0.02 and \$0.06 for training a CNN, a RNN, and Multilayer NN, respectively. The feasible space is represented in Table 7.1. For the CNN, there are 111 feasible configurations. However, only 69 configurations (23.96% of the search space for $s = 1$) respect the cost constraint and have a high accuracy (i.e., accuracy higher than 85%). Also, 37.85% of the configurations have high accuracy but do not meet the constraint. For training a RNN, the number of feasible configurations is 178 representing 61.8% of the space for $s = 1$. However, only 14.58% of the configurations are feasible and have high accuracy. To train a Multilayer NN, 55.8% of the configurations respect the cost constraint using the full dataset. 27.28% of the configurations have high accuracy, but only 8.68% of configurations are feasible and have high accuracy. All theses experiments were executed in machines equipped with an Intel Xeon Platinum 8176 Processor and 256GB of memory running the operating system Linux Ubuntu 18.04 LTS with an architecture x84_64.

7.3 Impact of Subsampling on Cost of Optimization Process

This section evaluates Fabulinus and its ability to find the optimal configuration that maximizes the performance and ensures the cost constraints for each job. Therefore, each job is evaluated on exploration cost, time, and number of explorations.

Fabulinus is compared with the Fabolas approach. Fabulinus extends the acquisition function of Fabolas in order to incorporate constraints. The proposed system is also compared with standard BO approaches using the $\mathrm{EI_C}$ and the EI. Since Fabolas and BO with EI do not take into account constraints, it is expected that the metric used to evaluate the systems, i.e., the $Accuracy_C$ will be zero, and these two approaches will recommend configurations that do not respect the constraint. On the other hand, since Fabulinus and the standard BO with $\mathrm{EI_C}$ take into account constraints, it is expected that the recommend configurations have an $Accuracy_C$ equals to the value of the accuracy, i.e., the recommended configurations meet the cost constraint.

7.3.1 CNN Dataset

Analyzing Figure 7.1(a), it is possible to see that Fabulinus identifies solutions with a given value of $Accuracy_C$ at a fraction of the time and cost required by the other considered methods. Fabulinus can recommend configurations that have high accuracy and ensures the cost constraint, unlike Fabolas that always outputs configurations that do not respect the constraint. Therefore, the proposed acquisition function used by Fabulinus (Equation 4.4) can select configurations to evaluate that increase the knowledge about the optimum and have a high probability to meet the constraint.

Although the number of explorations required by Fabulinus to build the initial models is higher than standard BO approaches without subsampling (the instant when the initial sampling ends is marked in the figures with a circle), the first spends less money to create the model. Fabulinus reduces the cost of the initial sampling phase by approximately 3%. This happens because Fabulinus evaluates cheaper configurations in subsampled datasets, unlike standard BO approaches without subsampling. Fabulinus reduces the cost of the optimization process in 35,52%, 58.34%, and 88.24% comparing with Fabolas, BO using $\mathrm{EI_C}$ and EI, respectively.

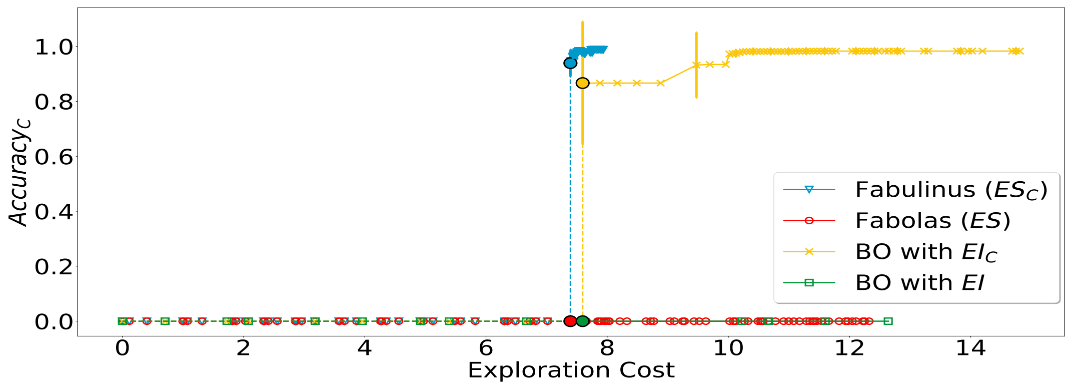
Since the exploration time and cost are dependent, there is a similar behavior for the exploration time (Figure 7.1(b)). However, if the overhead time necessary to compute the acquisition function is analyzed (Figure 7.1(d)), the optimization time, i.e., the exploration time plus the overhead, is higher for Fabulinus. To train a CNN, the training time to evaluate one configuration is in the same order of magnitude of the overhead time. So, none of them can be neglected. Fabulinus consumes 1.06 more time than BO using the $\mathrm{EI_C}$. When the model starts to be used, the computation of the acquisition function to recommend a configuration to sample takes approximately 6 minutes. The overhead time increases as there are more sampled configurations. At the end of exploration, after evaluated one hundred configurations, Fabulinus spends more than 12 minutes on average to recommend a configuration. However, using a very small number of iterations, the overhead time does not have a significant impact on the time consumed, and Fabulinus can recommend configurations with high accuracy that meet the cost constraint.

7.3.2 RNN Dataset

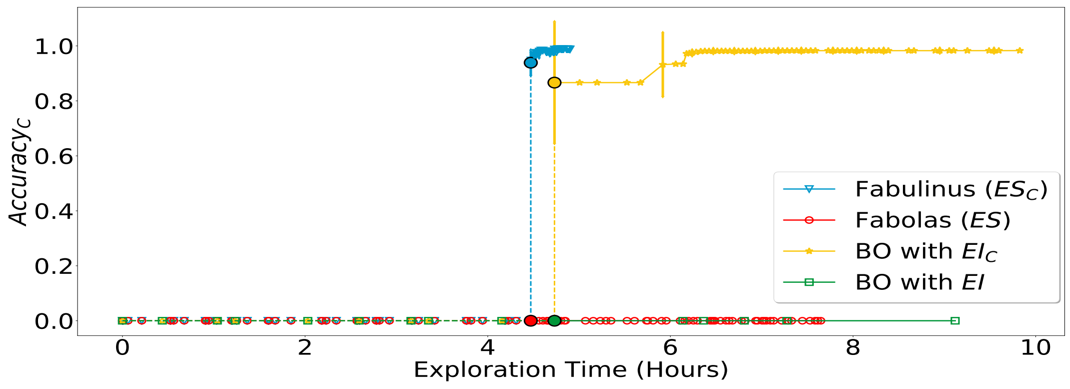
Next, the four optimizers were evaluated for training a RNN. The results are shown in Figure 7.2. Fabulinus and BO with $\mathrm{EI_C}$ always recommend configurations that have high accuracy and ensures the cost constraint, unlike Fabolas that always outputs configurations that do not respect the constraint. The incumbent configuration predicted by Fabolas never respects the constraints.

Similar to the previous job, using subsampling can reduce the cost paid in the initial sampling. In this case, there is Fabulinus reduces in exploration cost of the initial sampling phase by almost 9%. Fabulinus reduces in 40%, 80% and 79.7% the optimization cost comparing with Fabolas, BO using $\mathrm{EI_C}$ and EI.

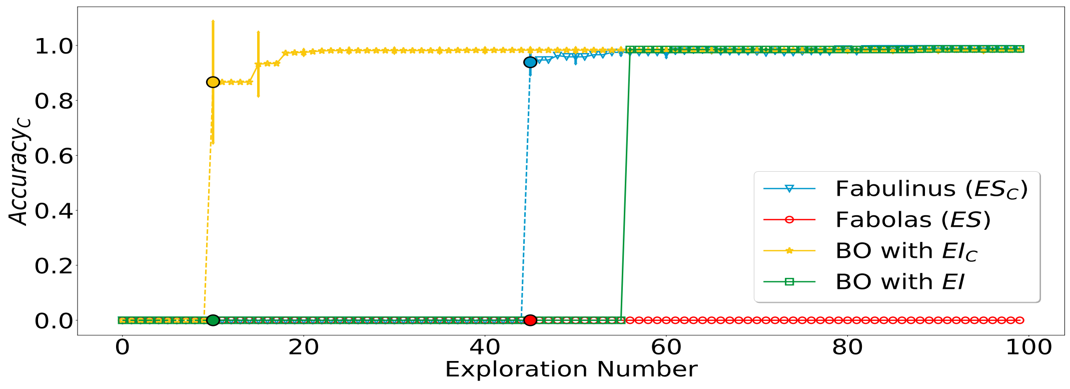
When the overhead is taken into account, the time that Fabulinus spends to recommend good configurations is 4.71, 6.46, and 12.4 times higher than the optimization time taken by Fabolas, BO with $\mathrm{EI_C}$ and EI, respectively.



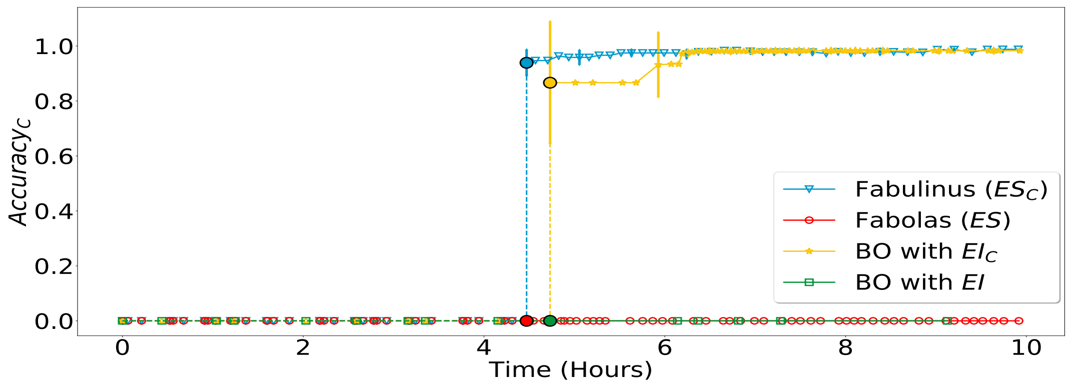
(a) Exploration Cost



(b) Exploration Time

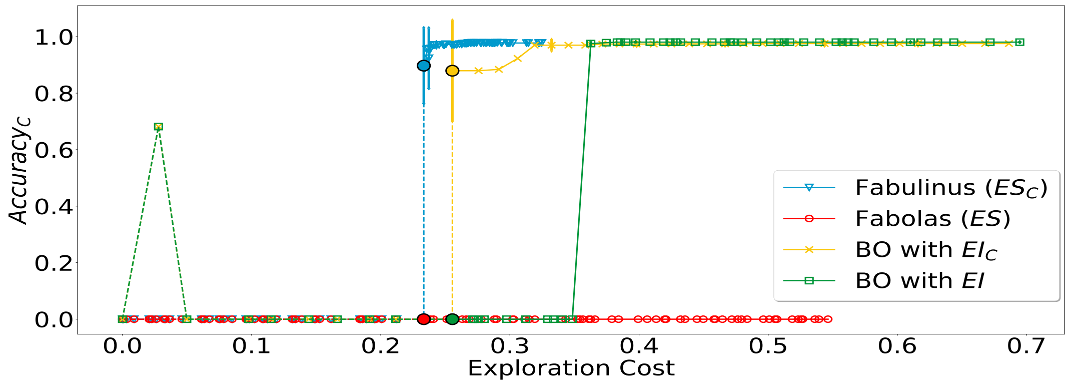


(c) Exploration Number

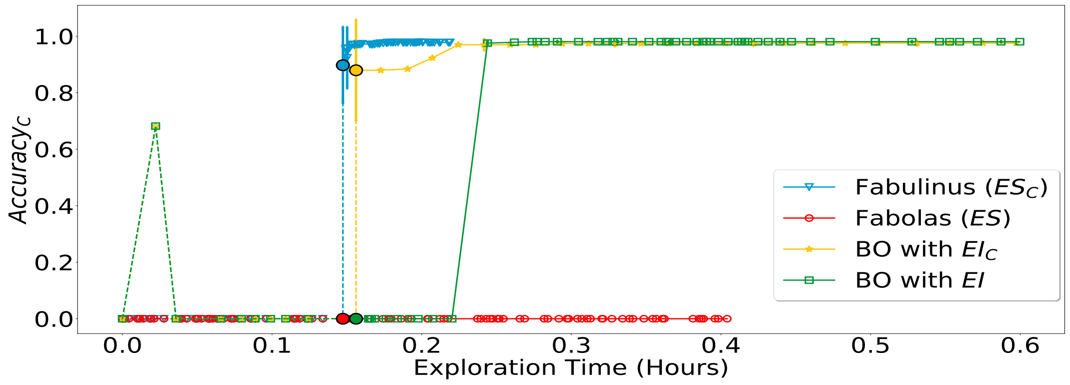


(d) Time (exploration time + overhead)

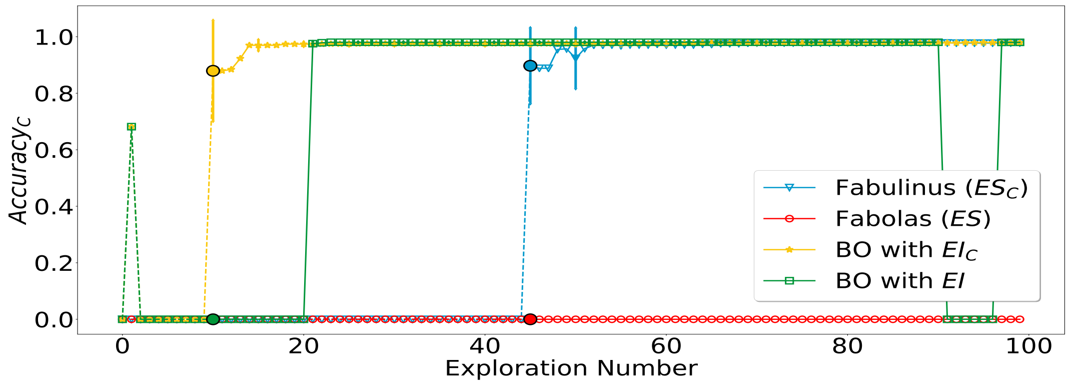
Figure 7.1: Optimization Process to train a CNN



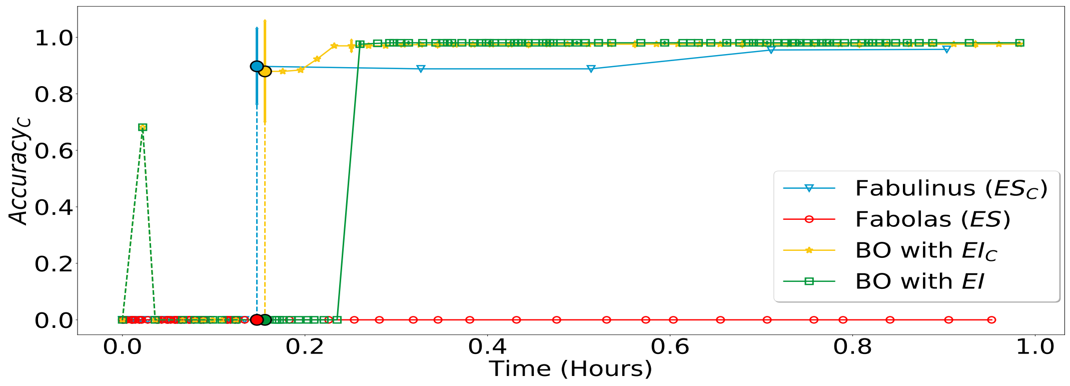
(a) Exploration Cost



(b) Exploration Time

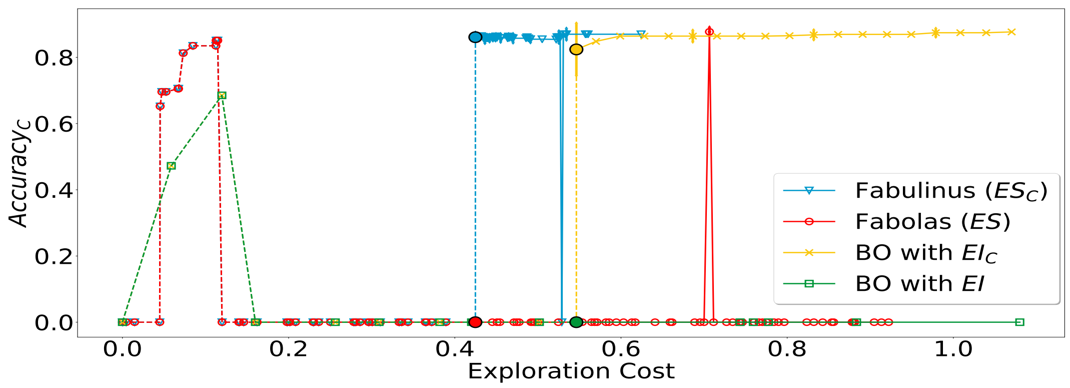


(c) Exploration Number

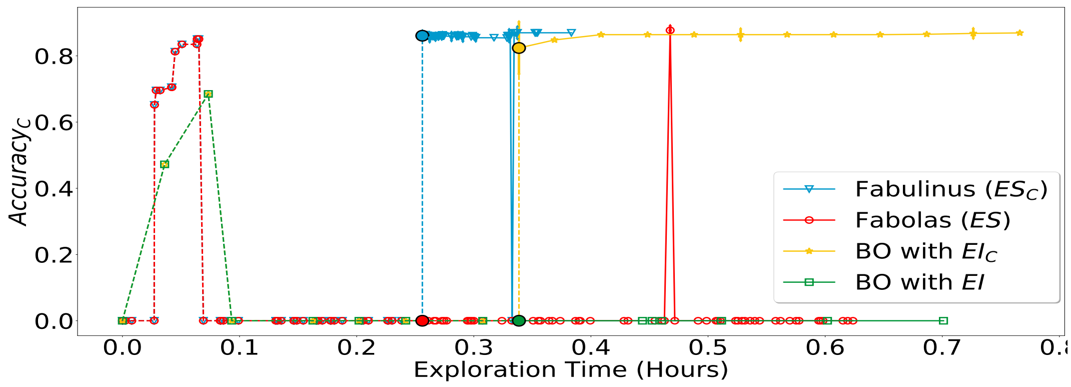


(d) Time (exploration time + overhead)

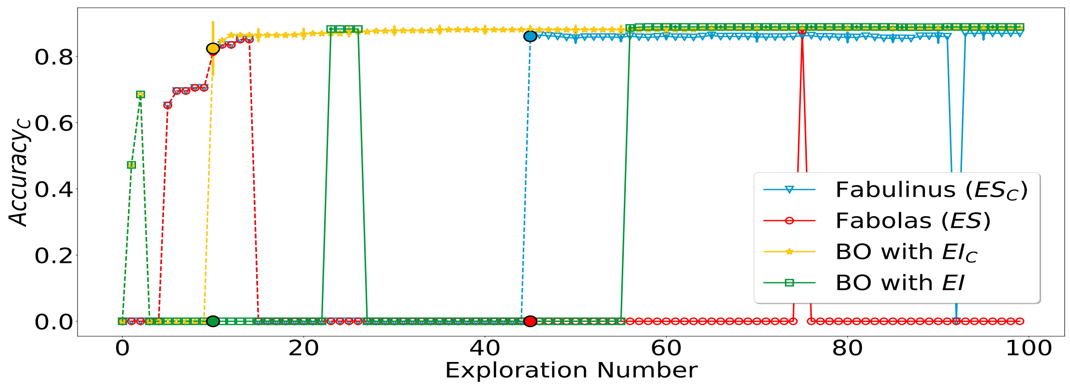
Figure 7.2: Optimization Process to train a RNN



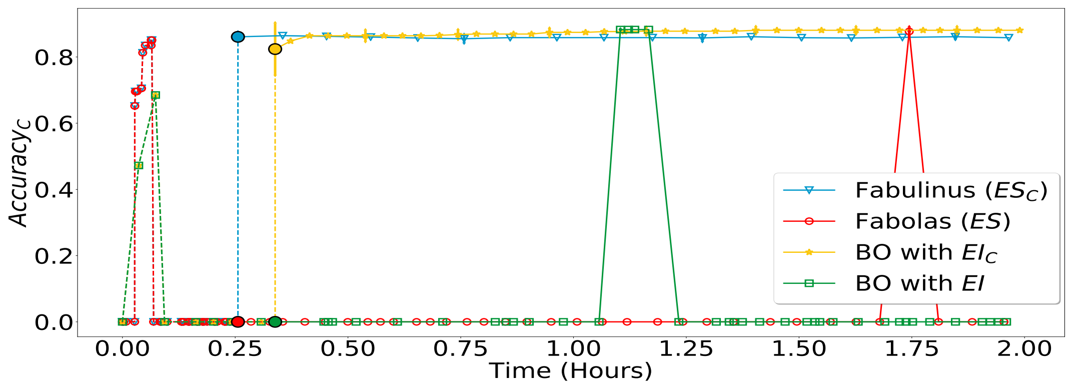
(a) Exploration Cost



(b) Exploration Time



(c) Exploration Number



(d) Time (exploration time + overhead)

Figure 7.3: Optimization Process to train a Multilayer NN

7.3.3 Multilayer NN Dataset

Figure 7.3 presents the evolution of the $Accuracy_C$ using Fabulinus, Fabolas, standard BO with EI_C, and EI. During the optimization process, there is a configuration recommended by Fabulinus that does not meet the constraint. Fabulinus only predicts the cost and accuracy of the incumbent and never runs it. Therefore, there will always have some uncertainty about the real performance and cost of the incumbent. However, in general, Fabulinus can recommend configurations that have high accuracy and complies with the constraint.

Fabulinus reduces the cost of the initial sampling by 22.2% when compared with standard BO without subsampling. For training a Multilayer NN, Fabulinus can also reduce the cost of the optimization process by 32.26%, 83.93%, and 86.6% comparing with Fabolas, BO with EI_C and EI, respectively. As in the previous case, Fabulinus explores more configurations but spends less money than BO approaches without subsampling. Fabulinus reduces the exploration time and recommends configurations with high accuracy the comply with the constraint.

Fabolas and BO using EI present a large number of recommendations that do not meet the constraint, as expected. Accounting the entire optimization time (i.e., exploration time plus overhead), Fabulinus spends 2.52 more time than Fabolas, 1.69 more time than BO with EI_C and 2.79% more time than BO with EI. To train a multilayer NN, usually, the execution time of a given configuration is shorter than the overhead required by Fabulinus. Notwithstanding, the first recommendations where the overhead does not have a considerable impact, Fabulinus recommends on average configurations that have high accuracy and meet the constraint.

Summary

This Chapter evaluates Fabulinus proposed in Chapter 4. Using the datasets gathered in this work, Fabulinus was compared with Fabolas, a standard BO using the EI_C and EI as an acquisition function. This evaluation tries to validate the acquisition function used by Fabulinus and proposed in this work. The results obtained for the training of the three networks show that it is always possible to reduce the cost of the optimization process. In particular, Fabulinus can reach a cost reduction of 58.34%, 83.93% and 80% comparing with a standard BO using the EI_C to train a CNN, a Multilayer NN and a RNN, respectively.

Through the use of subsampling, it is always possible to increase the number of evaluated configurations and reduce the cost spent in the initial sampling to construct the models. Therefore, when the models start to be used for predictions, the exploration cost and time are always lower using Fabulinus and Fabolas. Fabulinus presents a higher total optimization time, i.e., the exploration time plus the overhead necessary to compute the acquisition function and select the next configuration to evaluate. In the beginning, the overhead is around 6 minutes and increases with the number of evaluated configurations. In the end, the overhead can be higher than 12 minutes. However, the overhead does not have a significant and critical impact on the first predictions

To sum up, Fabulinus can reduce the cost of the optimization process through the use of subsampling a still recommend configurations that maximize the accuracy in the full dataset and comply with the constraint. The results in this section serve also to validate the effectiveness of the novel acquisition function employed by Fabulinus.

8 Conclusions and Future Work

This thesis propose two systems, Nephele and Fabulinus, that leverage subsampling techniques to optimize the training of machine learning models in the cloud.

Nephele aims at finding the best configuration that minimizes the training cost subject to user-defined constraints on accuracy, time budget for the optimization phase. Nephele exploits subsampling in order to reduce the training cost by adjusting the size of the dataset to trade-off, in a controlled way, the accuracy of the resulting models and the cost of the training process.

The results show that using subsampling, it is possible to reduce the cost by up to 75% comparing to state-of-the-art approaches, if one accepts an accuracy threshold of 85%. Further, despite the inclusion of the subsampling rate in the configuration space leads to an increase of the problem's dimensionality, the cost of Nephele' optimization process is comparable to, and often even lower than, that of equivalent BO-based methods that do not include sub-sampling in their configuration space.

The other proposed system, called Fabulinus, aims to find the configuration on the full dataset that maximizes the accuracy of ML jobs subject to a user-defined constraint on cost (or time), while evaluating configurations using only subsampled datasets which are usually cheaper and faster. The key novel contribution of Fabulinus is a new acquisition function, which selects the configuration and dataset size to test by keeping into account two factors: i) maximizing information on the loss-minimizing configuration on the full dataset per unit cost spent testing configurations, using recently proposed techniques in the transfer learning literature; ii) maximizing the likelihood that the cost constraint will be met by the recommended configuration, using the full dataset, based solely on information gathered by using subsampled datasets. We show that Fabulinus can reduce the optimization cost by a factor up to 6.6$\times$ when compared to classic BO-techniques that do not use subsampling, while effectively enforcing the specified cost constraints, unlike recent state-of-the-art techniques that use sub-sampling.

This work has opened a number of research question that would be interesting to address in future work:

- As for future research directions, Nephele can be incorporated with the Hyperband method in order to identify and drop under-performed configurations in the early phases of optimization. All these systems use Sequential model-based (SMBO) and are model-dependent to select the next configuration. The initial model constructed through randomly sampled configurations has a direct impact on the performance of the optimization. In particular, this approach is not efficient since it can evaluate expensive configurations reducing the available budget significantly. Furthermore, this random search can evaluate infeasible regions and miss evaluating relevant regions of the configuration space.

- Fabulinus could be extended to incorporate a budget for exploration as Nephele. Also, it would be interesting to integrate lookahead techniques in Fabulinus. The main challenge to pursue this goal is that the complexity to compute the acquisition function will increase significantly. This is problematic and may be not feasible in practice since Fabulinus has already large computational demands: in our experiments Fabulinus takes approximately six minutes to select a configuration at the beginning of optimization, and this time increases as more information becomes available. A possibility might be to use other recently proposed approximations, e.g., [37], to speed up the computation of the acquisition function.

- The acquisition function proposed in Fabulinus, the constrained Entropy Search (ES_C), may be modified in order to test alternative variants of this acquisition function. This is something that was not possible to test during this dissertation given the large amount of time needed to run statically meaningful tests with Fabulinus. For example, it would be interesting to test a variant of ES_C that simply multiplies the Entropy Search by the probability that a constraint is met without including in the dataset the predictions of the model in its current state. Such a test would allow to prove experimentally the relevance of this specific design choice of Fabulinus.
- Finally, it may be possible to extend the proposed systems in order to use other measurements like CPU usage, memory availability, or network monitoring to increase the quality of model predictions.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "Tensorflow: a system for large-scale machine learning". In: *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*. Vol. 16. Savannah, GA, USA, 2016, pp. 265–283.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming 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*. Boston, MA, USA, 2017, pp. 469–482.

[3] Amazon. *Amazon Elastic Compute Cloud (EC2)*. `https://aws.amazon.com/ec2/`.

[4] Peter Auer. "Using confidence bounds for exploitation-exploration trade-offs". In: *Journal of Machine Learning Research*. Vol. 3. Nov. JMLR, inc. 2002, pp. 397–422.

[5] Richard Bellman. "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515.

[6] Shai Ben-David and Reba Schuller Borbely. "Exploiting Task Relatedness for Mulitple Task Learning". In: *Conference on Learning Theory*. 2003.

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. Granada, Spain, 2011, pp. 2546–2554.

[8] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. Granada, Spain: Curran Associates Inc., 2011, pp. 2546–2554.

[9] Dimitri P Bertsekas. *Dynamic programming and optimal control*. Vol. 1. Athena Scientific Belmont, Massachusetts, 1996.

[10] L. Bottou, F. E. Curtis, and J. Nocedal. "Optimization Methods for Large-Scale Machine Learning". In: *CoRR* abs/1606.04838 (2016).

[11] Leo Breiman. "Bagging Predictors". In: *Machine Learning* 24.2 (1996), pp. 123–140.

[12] Leo Breiman. *Classification and regression trees*. Routledge, 1984.

[13] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[14] Leonard A. Breslow and David W. Aha. "Simplifying decision trees: A survey". In: *The Knowledge Engineering Review* 12.1 (1997), pp. 1–40.

[15] Thomas M Breuel. "Benchmarking of LSTM networks". In: *CoRR* abs/1508.02774 (2015).

[16] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: *CoRR* abs/1012.2599 (2010).

[17] Eric Brochu, Matthew Hoffman, and Nando de Freitas. “Portfolio Allocation for Bayesian Optimization”. In: *CoRR* abs/1009.5419 (2011).

[18] Maria Casimiro. “Lynceus: Long-Sighted, Budget-Aware Online Tuning of Cloud Applications”. MA thesis. Lisbon, Portugal: Instituto Superior Técnico, 2018.

[19] Maria Casimiro, Diego Didona, Paolo Romano, Luís Rodrigues, and Willy Zwanepoel. “Lynceus: Tuning and Provisioning Data Analytic Jobs on a Budget”. In: *CoRR* abs/1905.02119 (2019).

[20] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. “Deep Big Multilayer Perceptrons for Digit Recognition”. In: *Neural Networks: Tricks of the Trade: Second Edition*. Springer Berlin Heidelberg, 2012, pp. 581–598.

[21] D. D. Cox and S. John. “A statistical method for global optimization”. In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 2. 1992, pp. 1241–1246.

[22] Christina Delimitrou and Christos Kozyrakis. “HCloud: Resource-Efficient Provisioning in Shared Cloud Systems”. In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 2016, pp. 473–488.

[23] Christina Delimitrou and Christos 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*. Salt Lake City, UT, USA, 2014, pp. 127–144.

[24] Li Deng. “The MNIST database of handwritten digit images for machine learning research [best of the web]”. In: *IEEE Signal Processing Magazine*. Vol. 29. 6. IEEE. 2012, pp. 141–142.

[25] Michael D Ekstrand, John T Riedl, and Joseph A Konstan. “Collaborative filtering recommender systems”. In: *Foundations and Trends in Human–Computer Interaction*. Vol. 4. 2. Now Publishers, Inc. 2011, pp. 81–173.

[26] Stefan Falkner, Aaron Klein, and Frank Hutter. “BOHB: Robust and Efficient Hyperparameter Optimization at Scale”. In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. 2018, pp. 1437–1446.

[27] Peter I. Frazier. “A Tutorial on Bayesian Optimization”. In: *CoRR* abs/1807.02811 (2018).

[28] Jacob R. Gardner, Matt J. Kusner, Zhixiang Xu, Kilian Q. Weinberger, and John P. Cunningham. “Bayesian Optimization with Inequality Constraints”. In: *Proceedings of the 31st International Conference on Machine Learning*. Vol. 32. Beijing, China, 2014, pp. 937–945.

[29] David Ginsbourger and Rodolphe Le Riche. “Towards GP-based optimization with finite time horizon”. In: *Proceedings of the 9th International Workshop in Model-Oriented Design and Analysis*. Bertinoro, Italy, 2010, pp. 89–96.

[30] Javier González, Michael Osborne, and Neil D. Lawrence. “GLASSES: Relieving The Myopia Of Bayesian Optimisationn”. In: *CoRR* abs/1510.06299 (2015).

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[32] Google. *Google Compute Engine*. `https://cloud.google.com/compute/`.

[33] Google Compute Engine. *Custom Machine Types*. `https://cloud.google.com/compute/docs/machine-types#custom_machine_types`. (Visited on 09/04/2019).

[34] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. “The WEKA Data Mining Software: An Update”. In: *ACM SIGKDD explorations newsletter*. Vol. 11. 1. 2009, pp. 10–18.

[35] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. "Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets". In: *Proceedings of the 12th European Conference on Computer Systems*. Belgrade, Serbia, 2017, pp. 589–604.

[36] Philipp Henning and Christian J. Schuler. "Entropy Search for Information-Efficient Global Optimization". In: *Journal of Machine Learning Research*. Vol. 13. Jan. JMLR, inc. 2012, pp. 1809–1837.

[37] José Miguel Hernández-Lobato, Matthew W. Hoffman, and Zoubin Ghahramani. "Predictive entropy search for efficient global optimization of black-box functions". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems*. Vol. 1. Montreal, Canada, 2014, pp. 918–926.

[38] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. "Scout: An Experienced Guide to Find the Best Cloud Configuration". In: *CoRR* abs/1803.01296 (2018).

[39] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-based Optimization for General Algorithm Configuration". In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. Rome, Italy, 2011, pp. 507–523.

[40] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin Murphy. "Time-bounded Sequential Parameter Optimization". In: *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*. Venice, Italy, 2010, pp. 281–298.

[41] Peter Jaeckel. *A note on multivariate Gauss-Hermite quadrature*. `https://pdfs.semanticscholar.org/0e39/411d.pdf`. 2005.

[42] Kevin Jamieson and Ameet Talwalkar. "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Vol. 51. Proceedings of Machine Learning Research. PMLR, 2016, pp. 240–248.

[43] Donald R. Jones. "A Taxonomy of Global Optimization Methods Based on Response Surfaces". In: *Journal of Global Optimization*. Vol. 21. 4. Springer. 2001, pp. 345–383.

[44] Donald R. Jones, Matthias Schonlau, and William J. Welch. "Efficient Global Optimization of Expensive Black-Box Functions". In: *Journal of Global Optimization*. Vol. 13. 4. Springer. 1998, pp. 455–492.

[45] Bogumił Kamiński, Michał Jakubczyk, and Przemysław Szufel. "A framework for sensitivity analysis of decision trees". In: *Central European Journal of Operations Research* 26.1 (2018), pp. 135–159.

[46] J. Kiefer and J. Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function". In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[47] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *CoRR* abs/1412.6980 (2014).

[48] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. "Fast Bayesian Optimization of MachineLearning Hyperparamaters on Large Datasets". In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Vol. 54. PMLR, 2017, pp. 528–536.

[49] Tammo Krueger, Danny Panknin, and Mikio Braun. "Fast cross-validation via sequential testing". In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1103–1155.

[50] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics*. Vol. 22. 1. 1951, pp. 79–86.

[51] H. J. Kushner. “A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise”. In: *Journal of Basic Engineering*. Vol. 86. 1. The American Society of Mechanical Engineers. 1964, pp. 97–106.

[52] Remi R. Lam, Karen E. Willcox, and David H. Wolpert. “Bayesian Optimization with a Finite Budget: An Approximate Dynamic Programming Approach”. In: *Proceedings of the 29th Neural Information Processing Systems Conference*. Barcelona, Spain, 2016, pp. 883–891.

[53] Remi Lam and Karen Willcox. “Lookahead Bayesian Optimization with Inequality Constraints”. In: *Proceedings of the 30th Neural Information Processing Systems Conference*. Long Beach, CA, USA, 2017, pp. 1890–1900.

[54] L. Li, K. Jamieson, Giulia DeSalvo, A. Rostamizadeh, and A. Talwalkar. “Hyperband: A novel bandit-based approach to hyperparameter optimization”. In: *Journal of Machine Learning Research* 18 (2018), pp. 1–52.

[55] Qing Liu and Donald A Pierce. “A note on Gauss—Hermite quadrature”. In: *Biometrika*. Vol. 81. 3. Oxford University Press, Biometrika Trust. 1994, pp. 624–629.

[56] Daniel Lizotte. “Practical Bayesian Optimization”. PhD thesis. University of Alberta, Canada, 2018.

[57] Radford M. Neal. “Monte Carlo Implementation of Gaussian Process Models for Bayesian Regression and Classification”. In: *CoRR* abs/physics/9701026 (1997).

[58] Bertil Matérn. *Spatial Variation*. Berlin, Germany: Springer-Verlag, 1986.

[59] Peter M. Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011.

[60] Microsoft Azure. *Virtual Machines*. `https://azure.microsoft.com/en-us/services/virtual-machines/`.

[61] John Mingers. “An empirical comparison of selection measures for decision-tree induction”. In: *Machine learning*. Vol. 3. 4. Springer. 1989, pp. 319–342.

[62] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, 1997.

[63] J Mockus, Vytautas Tiešis, and Antanas Žilinskas. “The Application of Bayesian Methods for Seeking the Extremum”. In: *Toward Global Optimization*. Vol. 2. Elsevier. 1978, pp. 117–128.

[64] Iain Murray and Ryan Prescott Adams. “Slice sampling covariance hyperparameters of latent Gaussian models”. In: *Proceedings of the 23rd International Conference on Neural Information Processing Systems)*. Vol. 2. Vancouver, British Columbia, Canada, 2010, pp. 1732–1740.

[65] Michael A. Osborne, Roman Garnett, and Stephen J. Roberts. “Gaussian processes for global optimization”. In: *3rd International Conference on Learning and Intelligent Optimization (LION3)*. 2009, pp. 1–15.

[66] Martin Pelikan, David E. Goldberg, and Erick Cantú-Paz. “BOA: The Bayesian Optimization Algorithm”. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*. Vol. 1. Orlando, FL, USA, 1999, pp. 525–532.

[67] Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Vol. 703. John Wiley & Sons, 2007.

[68] J. R. Quinlan. “Simplifying decision trees”. In: *nternational Journal of Man-Machine Studies - Special Issue: Knowledge Acquisition for Knowledge-based Systems. Part 5* 27.3 (1987), pp. 221–234.

[69] J. Ross Quinlan. “Induction of decision trees”. In: *Machine learning*. Vol. 1. 1. 1986, pp. 81–106.

[70] Martin Raab and Angelika Steger. “”Balls into Bins” - A Simple and Tight Analysis”. In: *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*. Berlin, Germany: Springer-Verlag, 1998, pp. 159–170.

[71] Laura Elena Raileanu and Kilian Stoffel. “Theoretical comparison between the gini index and information gain criteria”. In: *Annals of Mathematics and Artificial Intelligence*. Vol. 41. 1. Springer. 2004, pp. 77–93.

[72] Carl Edward Rasmussen and Malte Kuss. “Gaussian Processes in Reinforcement Learning”. In: *Proceedings of the 16th Neural Information Processing Systems Conference*. Whistler, British Columbia, Canada, 2003.

[73] Carl Edward Rasmussen and Christopher K.I. Williams. *Gaussian Processes for Machine Learning*. Cambridge, MA, USA: MIT Press, 2006.

[74] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[75] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. “Collaborative Filtering Recommender Systems”. In: *The Adaptive Web: Methods and Strategies of Web Personalization*. Springer Berlin Heidelberg, 2007, pp. 291–324.

[76] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*. Vol. 2. Lake Tahoe, Nv, USA, 2012, pp. 2951–2959.

[77] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. “Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design”. In: *CoRR* abs/0912.3995 (2010).

[78] Michael Stein. “Large Sample Properties of Simulations Using Latin Hypercube Sampling”. In: *Technometrics*. Vol. 29. 2. Taylor & Francis, Ltd., American Statistical Association, American Society for Quality. 1987, pp. 143–151.

[79] Michael L. Stein. *Interpolation of Spatial Data: Some Theory for Kriging*. NewYork, NY, USA: Springer-Verlag, 1999.

[80] James Stone. *Bayes’ Rule: A Tutorial Introduction to Bayesian Analysis*. June 2013.

[81] Kevin Swersky, Jasper Snoek, and Ryan P. Adams. “Multi-task Bayesian Optimization”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems*. Vol. 2. Lake Tahoe, Nv, USA, 2013, pp. 2004–2012.

[82] Michalis K. Titsias, Neil D. Lawrence, and Magnus Rattray. “Efficient sampling for Gaussian Process inference using control variables”. In: *Proceedings of the 21st International Conference on Neural Information Processing Systems*. Vancouver, British Columbia, Canada, 2008, pp. 1681–1688.

[83] Aimo Torn and Antanas Zilinskas. *Global Optimization*. Springer-Verlag, 1989.

[84] Lisa Torrey and Jude W. Shavlik. “ransfer Learning”. In: *Handbook of Research on Machine Learning Applications*. 2009.

[85] Christopher K.I. Williams and Carl Edward Rasmussen. “Gaussian processes for regression”. In: *Advances in neural information processing systems*. 1996, pp. 514–520.

[86] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. “Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach”. In: *Proceedings of the 8th ACM Symposium on Cloud Computing*. Santa Clara, CA, USA, 2017, pp. 452–465.

[87] Dani Yogatama and Gideon Mann. “Efficient Transfer Learning Method for Automatic Hyperparameter Tuning”. In: *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics)*. Vol. 33. 2014, pp. 1077–1085.

Appendix A

This section details the implementation of the stop condition defined to bring the training of the neural networks (NNs) (see Chapter 5).

In order to prevent the job from running forever in suboptimal configurations that may take an unknown time to reach a given accuracy constraint or that may even not reach such accuracy. To accomplish this, it is assumed that using a large portion of the dataset to train the NN should be enough to allow optimal and near-optimal configurations to achieve acceptable accuracies. In this work, this large portion consists of 90% of the dataset and this threshold is controlled by monitoring the number of iterations performed. In each iteration, each worker selects at random a batch of images (of 16 or 256 images) to train the network. At the end of each iteration, each worker sends the results to the supervisor (called *Parameter Server*) that coordinates the training process. The *Parameter Server* calculates the new weights that minimize the loss function, and it sends the updates to the workers.

The batch of images is created through random selection among the dataset. Determining the number of iterations required to ensure that 90% of the dataset has been evaluated creates a problem commonly designated ”Balls into bins problem” [70]. In this problem, there are m balls and n boxes (or bins). In each step, m balls are placed randomly into m different bins.

There are two possible cases to be considered: either a bin is empty or a bin has at least one ball. These two cases can be modeled by the random variable X_i. When a bin has at least one ball, $X_i = 1$. When a bin is empty, $X_i = 0$. This corresponds to having X_i follow a Bernoulli distribution. For each bin, the probability that it has one or more balls after throwing m balls is equal to

$$\begin{gathered} P(\text{bin has at least one ball}) = 1 - P(\text{bin does not have balls}) \\ P(X_i = 1) = 1 - P(X_i = 0) = 1 - \left(\tfrac{n-1}{n}\right)^m . \end{gathered} \tag{1}$$

The expected value of a Bernoulli random variable X_i is

$$\mathbb{E}(X_i) = P(X_i = 1) \cdot 1 + P(X_i = 0) \cdot 0 = P(X_i = 1) = 1 - \left(\frac{n-1}{n}\right)^m . \tag{2}$$

Let distribution $Y = X_1 + X_2 + \cdots + X_N$ be the number of bins with at least one ball. The expected value of Y is

$$\mathbb{E}(Y) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_n) = n \cdot \mathbb{E}(X_i) = n \cdot \left(1 - \left(\frac{n-1}{n}\right)^m\right) \tag{3}$$

It is possible to formulate the initial problem of determining the number of iterations I required to ensure that at least 90% of the dataset is used to train the NN as a ”Balls into bins” problem. In order to achieve this, one must consider the dataset size S to be the number of bins and the product of the number of iterations I and of the batch size b to be the number of balls. Thus, the probability of an imagine being selected at least once is given by Equation 5.4.

$$P(\text{image is selected at least once}) = 1 - P(\text{image is never selected}) = 1 - \left(\frac{S-1}{S}\right)^{Ib}. \tag{4}$$

Let $X_i = 1$, if an image i is selected at least once and, $X_i = 0$, otherwise. The expected value of X_i is $\mathbb{E}(X_i) = 1 - \left(\frac{S-1}{S}\right)^{Ib}$. Also, let Y be the total number of images used for training. Thus, the expected value of Y is $\mathbb{E}(Y) = S \cdot \mathbb{E}(X_i) = S \cdot \left(1 - \left(\frac{S-1}{S}\right)^{Ib}\right)$.

Therefore, to ensure that at least $\alpha\%$ of the data set is used, $\mathbb{E}(Y_i) \geq \alpha \cdot S$. Through mathematical manipulation, it is possible to determine the number of iterations required to use $\alpha\%$ of the dataset to train, as a function of dataset size S and batch size b.

$$\begin{aligned} \mathbb{E}(Y) \geq \alpha \cdot S \Leftrightarrow S \cdot \left(1 - \left(\frac{S-1}{S}\right)^{Ib}\right) \geq \alpha \cdot S \Leftrightarrow 1 - \alpha \geq \left(\frac{S-1}{S}\right)^{Ib} \Leftrightarrow \\ \Leftrightarrow \log(1-\alpha) \leq Ib \cdot \log\left(\frac{S-1}{S}\right) \Leftrightarrow I \geq \frac{\log(1-\alpha)}{b \cdot \log\left(\frac{S-1}{S}\right)} \end{aligned} \tag{5}$$

In this work, α was set to 0.9.