Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Knut Vågnes Eriksen

Hyperparameter Optimization for Neural Network-based Virtual Flow Metering

An industrial case study

Master's thesis in Cybernetics and Robotics
Supervisor: Bjarne Andre Grimstad
Co-supervisor: Lars Struen Imsland and Maurício Bezerra de Souza Júnior
June 2022

NTNU
Norwegian University of Science and Technology

Knut Vågnes Eriksen

Hyperparameter Optimization for Neural Network-based Virtual Flow Metering

An industrial case study

Master's thesis in Cybernetics and Robotics
Supervisor: Bjarne Andre Grimstad
Co-supervisor: Lars Struen Imsland and Maurício Bezerra de Souza Júnior
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Preface

This work is a continuation of the specialization project report done by the candidate in the course *TTK4550 Engineering Cybernetics, Specialization Project*.

This work, and the specialization report, is written in collaboration with the Norwegian company Solution Seeker AS(ORG:911 576 236). Solution Seeker provided access to their machine learning model, dataset, and cloud resources. A special thanks to Bruna, Dani, Gui, Kevinho, Kristoffer, Rafa, and Tor Istvan from Solution Seeker for taking time out of their schedule to discuss and aid with various small tasks to facilitate this project.

Først en stor takk til Bjarne, Lars og Mauricio for god veiledning og rask respons på det som har dukket opp gjennom denne prosessen.

Så en stor obrigado til Lars, Mauricio og Solution Seeker for muligheten til å skrive denne oppgaven fra Brasil, og spesielt Bruna, Dani, Gui, Kevinho, og Rafa for et flott opphold i Rio de Janeiro.

Til slutt en takk til alle som har støttet, muntret og utfordret meg gjennom disse tjuefire-og-et-halvt årene.

Saúde!

Sammendrag

Hyperparametere er parameterne som kontrollerer hvordan en maskinlæringsalgoritme lærer, og hyperparameteroptimalisering er prosessen med å optimalisere disse parameterne. Riktig valg av hyperparameter verdier er avgjørende for å oppnå tilfredsstillende resultater, og optimalisering av dem kan heve ytelsen til en algoritme betydelig. Dessverre finnes det ingen generell hyperparameteroptimaliseringsalgoritme som alltid presterer bedre enn de andre. Solution Seeker har utviklet en datadrevet virtuell strømningsmåler, en maskinlæringsalgoritme som estimerer den totale strømmen av vann, olje og gass gjennom et rør. Dette arbeidet identifiserer egnede hyperparameteroptimaliseringsalgoritmer håndtilpasset denne virtuelle flytmåleren, og gir Solution Seeker en bedriftsspesifikk guide til hyperparameteroptimalisering, for forhåpentligvis å danne et konkurransefortrinn for virksomheten.

Interessante algoritmer er identifisert gjennom en litteraturgjennomgang av eksisterende hyperparameteroptimaliseringsalgoritmer sentrert rundt dype nevrale nettverk. Egnetheten til disse algoritmene blir deretter diskutert i sammenheng med den virtuelle strømningsmåleren gjennom en omfattende case studie om hyperparameteroptimalisering av den virtuell strømningsmåleren innenfor et begrenset tids- og kostnadsbudsjett. Resultatene fra case studiet støtter bruken av en modellbasert hyperparameteroptimaliseringsalgoritme, muligens i kombinasjon med en enkel multifidelity-teknikk.

Abstract

Hyperparameters are the parameters that control how a machine learning algorithm learns, and *Hyperparameter Optimization* (HPO) is the process of optimizing these parameters. Appropriately selecting the values of hyperparameters is paramount for achieving satisfying results, and optimizing them can significantly excel the performance of an algorithm. Unfortunately, there exists no general HPO-algorithm that outperforms the other. Solution Seeker has developed a data-driven *virtual flow meter* (VFM), a machine learning algorithm that estimates the total flow of water, oil & gas through a pipe. This work identifies suitable HPO-algorithms hand-tailored to this VFM, and provides Solution Seeker with a company-specific guide to HPO, to hopefully help the business gain a competitive advantage.

Interesting algorithms are identified through a literature review of existing HPO-algorithms centred around deep neural networks. The fitness of these algorithms is then discussed in the context of the VFM through an extensive case study on hyperparameter optimization of the VFM within a restricted time and cost budget. The case study results support the use of a model-based HPO-algorithm, possibly in combination with a simple multi-fidelity (MF) technique.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Source Code

Nomenclature

Abbreviations

AutoML	Automated Machine Learning
BGD	Batch Gradient Descent
BO	Bayesian Optimization
BOHB	Bayesian Optimization Hyperband
CI	Confidence Interval
EI	Expected Improvement
GA	Genetic Algorithm
GP	Gaussian Process
GS	Grid Search
HPO	Hyperparameter Optimization
KDE	Kernel Density Estimation
MBGD	Mini-Batch Gradient Descent
MF	Multi-Fidelity
MSE	Mean Square Error
MSR	Median Stopping Rule
MTL	Multitask Learning
NAS	Neural Architecture Search
PSO	Particle Swarm Optimization
ReLU	Rectified Linear Unit
RS	Random Search
SGD	Stochastich Gradient Descent
SMAC	Sequential Model-Based Algorithm Configuaration
SMBO	Sequential Model-Based Optimization
TPE	Tree-structured Parzen Estimator
VFM	Virtual Flow Meter

Case legends

CHK	Choke opening
FGAS	Fraction of gas to total flow
FOIL	Fraction of oil to total flow
NC	NeuralCompass
PDC	Pressure downstream choke
PWH	Pressure Well Head
QGL	How much gas lift has been used while producing
QTOT	Total flow rate
TWH	Temperature Well Head

Symbols

α	Learning rate
β	Momentum parameter
$\boldsymbol{\Lambda}$	Overall hyperparameter configuration space
$\boldsymbol{\lambda}$	A vector of hyperparamters
$\boldsymbol{\lambda}^*$	The optimal vector of hyperparameters
κ	Parameter norm penalty weight
Λ_n	Domain of the n-th hyperperameter
$\mathcal{A}$	Machine learning algorithm
$\mathcal{A}_{\boldsymbol{\lambda}}$	Machine learning algorithm $\mathcal{A}$ instantiated with hyperparamters $\boldsymbol{\lambda}$
$\nabla_\theta J(\theta)$	Gradient of J with respect to θ
$\Omega(\theta)$	Parameter norm penalty function
a	Activation function
$V(\cdot)$	Validation loss
J	Cost function

1 | Introduction

Machine learning has recently been applied to many industrial applications and is achieving ever-increasing better results. Part of the success is due to the increased and simplified gathering of data, increased computational power and accelerated hardware, as well as developments in deep learning architectures. However, during the last couple of years, progress in state-of-the-art performance has shifted from being caused by development in learning algorithms to something that rather extracts optimal behaviour of existing algorithms: Hyperparameter optimization.

Hyperparameter optimization has traditionally been dominated by expert manual tuning. However, the recently highlighted importance of optimal hyperparameter configuration has sparked a great scientific interest in automating hyperparameter selection. Not only is HPO directly applicable to extract the best possible behaviour of an already industrially applied model, but it is also crucial to be able to fairly compare the performance of two different models, as a suboptimal set of hyperparameters would give a false impression of the model's ability.

A major challenge with automated hyperparameter optimization is that there exists no single ideal optimization algorithm, and different HPO-methods perform different depending on the learning algorithm being optimized, the size of the dataset, and the resources available. The selection of an adequate optimization algorithm is therefore paramount for achieving optimal results.

This work identifies promising HPO-algorithms for optimizing the hyperparameters of a deep residual neural network used for regression analysis, and compares their performance through an industrial case study to provide practitioners with insight and a custom guide on how to tune their models in production, and fairly evaluate models in development.

This work is a continuation of the candidate's work in the course *TTK4550 Engineering Cybernetics, Specialization Project*, where the candidate familiarized himself with hyperparameter optimization and different HPO software programs.

1.1 Motivation

Solution Seeker is a Norwegian deep-tech Software-as-a-Service company that uses artificial intelligence to utilize sensor data in a novel fashion. One of the tools they provide is a virtual flow meter, which is used to estimate the total flow of water, oil & gas through a production pipe.

Knowledge of flow rate is key to an optimal petroleum production process, but, much like hyperparameter optimization, no field, well or production process is similar. Therefore, flow-rate sensors are often, to some degree, specially engineered for the specific installation, rendering them as costly instruments. In addition, deep-sea instalments are in itself difficult, and repairments even so. The intricate physics of multiphase flow also requires simplifications and assumptions to allow for feasible measurements and calculations, and as Thorn et al. (2012) summarizes, 'measuring it[the flow rate] remains one of the greatest challenges of petroleum production'.

The poor generality and high cost of the instruments have therefore sparked interest in developing simpler ways of measuring the flow rate. A virtual flow meter is one of the proposed techniques to do so. A VFM is a machine learning algorithm, that utilizes historical data of flow rates and other measurements such as temperature and pressure, to fit a model that can estimate the flow rate. It can be used in both a historical context, to estimate flow rate in periods when other measurements for some reason were unavailable, or for live estimation during production.

Solution Seeker is one of the world's first businesses to provide a VFM as a Software-as-a-Service solution. However, as a developing business, time is a limited resource, and properly formalizing hyperparameter optimization has up until the start of this work not been given any proper resources. By allocating more resources to this topic, we believe that Solution Seeker can gain a competitive advantage by 1) Improving the performance of models already in production, 2) Aiding and speeding up the research on new models, and 3) Reducing time spent on manually tuning hyperparameters.

This work aims thus to advance the performance of the VFM, but also pave the way for a company-specific guide to hyperparameter optimization and build a robust foundation for it, which the company can continue to develop to accelerate the performance of existing and new models.

1.2 Objectives and research questions

The following objectives were defined to aid Solution Seeker with gaining a competitive advantage:

- The primary objective of this work is to identify promising HPO-algorithms for a deep residual neural network used for regression analysis, understand the theory of the identified methods, and perform an extensive case study of the methods on the VFM to acquire knowledge and insight into the suitability of the different methods.
- The secondary object of this work is to provide Solution Seeker with a company-specific guide to hyperparameter optimization.

To meet these objectives, the following research questions were established:

1. How do a restricted time and cost budget affect the selection of HPO-algorithm?
2. Which aspects of the VFM are especially interesting for selecting an appropriate HPO-algorithm?
3. Does the VFM have any particular hyperparameters, and if so, are they especially interesting to optimize?

1.3 Outline

Chapter 1 introduces this work. The motivation for the work is given in Section 1.1, and the objectives and research questions are stated in Section 1.2. Section 1.3 describes the outline of the report.

Chapter 2 provides the necessary background for this work. First, a literature review of existing HPO-algorithms for neural networks is given Section 2.1. The literature review is written in a conceptual way to promote understanding of the concepts of the different methods, rather than diving into their theoretical world. Then a brief note on the difficulties of comparing different HPO-algorithms is given in Section 2.2.

Chapter 3 explains the theory needed to understand the case study and analyse the results. In Section 3.1, the basic concepts of machine learning are introduced, before a more thorough explanation of neural networks is presented in Section 3.2. Finally, in Section 3.3, the problem of hyperparameter optimization is formally defined. Then, the theory behind a selection of HPO-algorithms is explained. At last, the section discusses two important aspects of HPO: The exploitation-exploration trade-off and the robustness of a hyperparameter configuration.

Chapter 4 describes the neural network used in the case. First, Section 4.1 provides some brief context for the case. In Section 4.2 the dataset used is described, before Section 4.3 describes the structure of the neural network, and Section 4.4 describes the training and regularization of the network.

Chapter 5 shows the method used to perform the case study, and forms a guide to HPO for Solution Seeker. In Section 5.1 interesting hyperparameters of the neural network are identified. Then, in Section 5.2, the selection of HPO-algorithms is explained. Section 5.3 describes how the experiment of the case was designed, along with the hypothesis for the experiment. In Section 5.4 it is described how the experiment was programmatically implemented. The code is made available for Solution Seeker, but not to the public due to proprietary reasons.

Chapter 6 presents the results of the experiment, and Chapter 7 discusses them. Finally, a conclusion, and suggestions for future works are given in Chapter 8.

2 | Background

2.1 Literature review

Many hyperparameter optimization algorithms have been proposed with different rates of success. This section gives a brief review of the most popular algorithms in use today and highlights some advantages and disadvantages of the methods. In addition, a brief review of multi-fidelity techniques and their connection to HPO is given. The algorithms chosen to be explored in-depth in this work were selected on the background of this review and the works referenced in it. For a more thorough explanation of the different methods, please see the referenced material. An overview of different HPO-algorithms and MF techniques is given in Figure 2.2.

Hyperparameter optimization algorithms generally have four desired qualities that are used to evaluate the performance (Falkner et al., 2018a; Feurer and Hutter, 2019):

1. **Strong Anytime Performance**: Since training neural networks is computationally exhaustive, both in terms of computational time and computational power, the optimization algorithm should be able to provide good hyperparameter configurations within a reasonably small budget.
2. **Strong Final Performance**: When the optimization algorithm is given a reasonably large budget, it should be converging towards a globally optimal solution.
3. **Scalability**: The algorithm should be able to handle a large search space, since a neural network has many hyperparameters. It should also be feasible for the algorithm to sample many configurations when training the neural network is inexpensive. Finally, the algorithm should also be able to utilize parallel resources.
4. **Flexibility**: Since hyperparameters can be, e.g., real-valued, integer, categorical, or conditional, the algorithm should be able to handle all types of these hyperparameter domains.

Model-free

Model-free optimization algorithms are algorithms that do not model the response of the objective function. *Grid search* (GS) is the most basic of these methods. It consists of defining a set of hyperparameters, and possible values for these hyperparameters, and then performing a full factorial experiment. Although simple, this

method is inefficient, and *Random search* (RS) was empirically, and theoretically, proven by Bergstra and Bengio (2012) to be more efficient for hyperparameter optimization than GS. Instead of performing a full factorial experiment as GS does, RS rather draws hyperparameters from a predefined probability distribution. This allows RS to effectively be able to search over more of the important hyperparameters than GS since not all hyperparameters have the same significance (Figure 2.1). Ever since this important contribution, random search has because of its simple non-assuming nature been used as the comparison ground for several state-of-the-art performing HPO-algorithms. Random search is in addition scalable since there is no connection between trials, and flexible since no assumptions are made, and can handle all types of hyperparameter domains. Both GS and RS will eventually find the optimal solution, as all possible configurations, in the end, will have been tested. However, arriving at this optimal solution may require many unnecessary evaluations, yielding a poor anytime performance and average final performance.

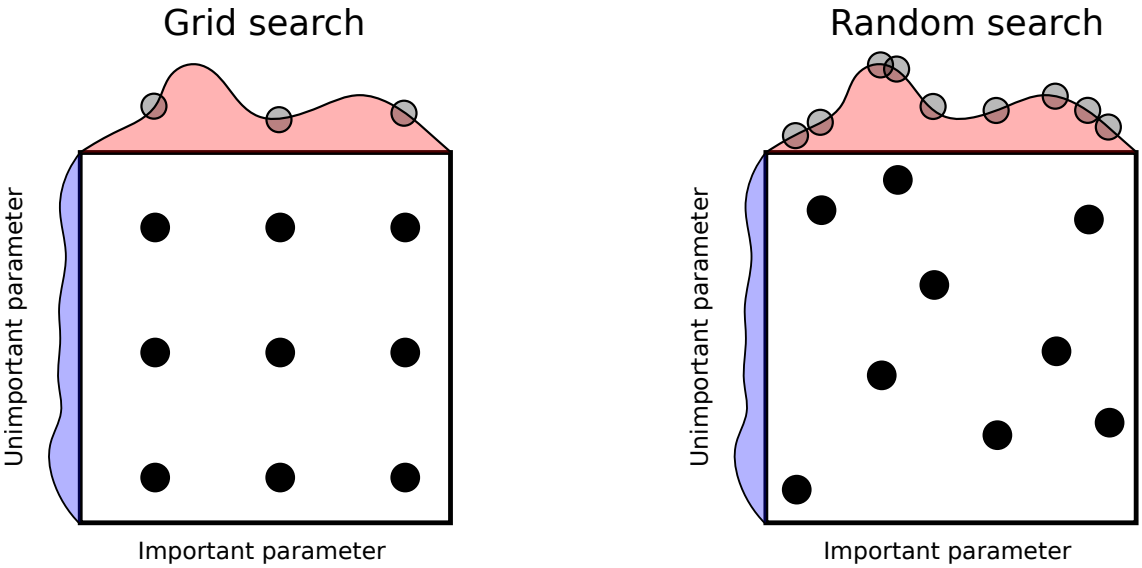


Figure 2.1: Grid Search vs Random Search. The illustration shows how random search effectively can sample more of the important hyperparameters. Adapted from Bergstra and Bengio (2012).

Another set of model-free HPO-algorithms is the *metaheuristic algorithms*. There exist various metaheuristic algorithms (see e.g., Glover and Kochenberger (2003) and Talbi (2009)), but particularly two methods are popular with HPO: 1) *Genetic Algorithm* (GA), and 2) *Particle Swarm Optimization* (PSO).

The genetic algorithm (Dan, 2013) is based on pairing individuals, dubbed chromosomes. A chromosome is a string, in which parts represent the hyperparameters, and which value is the candidate's value for the hyperparameter. The population of chromosomes is generated randomly, and then the individuals' fitness is evaluated before individuals are paired in favour of the fittest. The evaluation and pairing repeat until the algorithm reaches a converging criterion, or it times out. One advantage of GA is that it supports all types of hyperparameter domains. Its complexity is also average, scaling at a squared factor with the number of samples (Yang and Shami, 2020). In addition, GA is easy to parallelize (Feurer and Hutter, 2019). However,

GA introduces extra hyperparameters, such as the initial size of the population, how to calculate the fitness, and how to pair different individuals, rendering it less desirable for deep learning models that already have a large set of hyperparameters. There's also no guarantee for a strong anytime or final performance.

Particle swarm optimization (Kennedy and Eberhart, 1995) consists of a group (swarm) of individuals (particles). Each particle searches for the optimal solution by moving around in the search space. The movement of a particle is influenced by both its own best-known position, and the swarm's overall best-known position. The exchange of information is expected to move the swarm towards an optimal solution. PSO also supports all types of hyperparameter domains, and its complexity is low, scaling log-linear with the number of samples (Yang and Shami, 2020). It is also easy to parallelize (Feurer and Hutter, 2019). PSO does however require proper initialization to reach a global optimum (Yang and Shami, 2020), and as with GA, there is no guarantee for a strong anytime or final performance, although it empirically has achieved good results (Yang and Shami, 2020).

Model-based

None of the algorithms mentioned so far has come with any strong final performance guarantees. Model-based algorithms attempt to provide this desideratum by modelling the response surface of the objective function to make informed choices for where to sample next. These methods are specially designed for optimizing expensive black-box functions, such as a neural network. Particularly, the use of sequential model-based optimization (SMBO) algorithms (Hutter, Hoos et al., 2011) has been popularized and extensively researched during the last ten plus years. The authors used SMBO for general algorithm configuration and obtained state-of-the-art performance for the commercial mixed-integer programming solver CPLEX (Wikipedia contributors, 2022b), as well as the Boolean satisfiability problem (Wikipedia contributors, 2022a). An SMBO algorithm essentially consists of two main components: 1) A surrogate model to model the objective function, and 2) An acquisition function to determine where to sample next. Although there exist different acquisition functions, the main difference between SMBO algorithms lies with the surrogate model.

Bayesian optimization (BO) (Brochu et al., 2010; Shahriari et al., 2015) is one especially popular SMBO algorithm that has been used to obtain state-of-the-art performance on different datasets for deep learning applications (Snoek et al., 2012; Bergstra, Yamins et al., 2013; Mendoza et al., 2016). BO utilizes Bayes' theorem to model the objective function as a posterior distribution. There exist different ways of obtaining this posterior, but the three most popular used today are: 1) Gaussian processes, 2) Random forests, and 3) Tree-structured Parzen estimators.

Gaussian process (GP) (Williams and Rasmussen, 2006) is a standard surrogate model used in Bayesian optimization. A Gaussian process prior is used to model the posterior probability distribution of the response of the objective function to gain knowledge on where it would be wise to sample next. GPs have traditionally been popular because of their '*expressiveness, smooth and well-calibrated uncertainty*

estimates, and closed-form computability of the predictive distribution' (Feurer and Hutter, 2019). Although BO-GP is efficient and has shown strong final performance (Bergstra, Bardenet et al., 2011; Snoek et al., 2012; Li et al., 2017; Falkner et al., 2018a; Klein and Hutter, 2019), its application to hyperparameter optimization of deep learning models suffers from mainly two limitations: 1) It is not scalable, as it scales cubically with the number of samples, and 2) It is not flexible, as it assumes continuous real-valued hyperparameters, and does not natively support any other hyperparameter domains.

Random forests (Breiman, 2001) are a popular alternative surrogate model used with BO. Proposed by Hutter, Hoos et al. (2011) as *Sequential Model-Based Algorithm Configuration* (SMAC), it uses an ensemble of regression trees to model the objective function's posterior probability distribution. Therefore, SMAC can easily handle all hyperparameter domains. They also scale much better than GPs with the number of samples, scaling at a log-linear complexity due to the tree structure (Yang and Shami, 2020). The first versions of the AutoML frameworks Auto-WEKA (Thornton et al., 2013) and Auto-sklearn (Feurer, Klein et al., 2019) both used SMAC to obtain state-of-the-art AutoML performance. However, using random forests may have some undesired effects compared to GPs. Far away from sampled data, predictions made by random forests can be identical, which might result in a poor acquisition of new hyperparameter configurations. Gaussian processes, however, fall back on their prior, which by design produce more uncertain predictions, which again is more desirable in the exploitation-exploration trade-off context. Furthermore, the response surface obtained using random forests is discontinuous and non-differentiable, rendering the use of gradient-based optimization methods infeasible (Shahriari et al., 2015).

The final alternative surrogate model is the use of a Tree-structured Parzen Estimator (TPE), proposed by Bergstra, Bardenet et al. (2011). Instead of modelling the posterior distribution directly, as GPs and SMAC do, the TPE models two density functions for the likelihood of a hyperparameter being either good or bad. It does so by dividing the observations into two categories, good and bad, and then uses two simple one-dimensional Parzen estimators (kernel-density estimators) to model the two density functions. It models the density functions for each hyperparameter individually, and then selects hyperparameter configurations from a tree structure, which again allows the algorithm to handle all types of hyperparameter domains. In addition, the TPE also scales log-linear with the number of samples (Yang and Shami, 2020), and have shown great results from optimizing neural networks with large conditional configuration spaces (Feurer and Hutter, 2019).

Multi-fidelity

The three BO algorithms all have a strong final performance, but none of them provides a strong anytime performance. *Multi-fidelity optimization* is a set of techniques that tries to overcome this challenge by probing the performance of the neural network without using the entire resource pool. That is, these techniques try to evaluate the performance of a partially trained network. To determine what partially refers to, one must first define a budget. With neural networks, the budget is often

either only using a subset of the training data (Klein, Falkner et al., 2017), or training for the full amount of training iterations (Kandasamy et al., 2017). The idea is that by using low fidelity approximations, one can quickly approximate the final performance of the model to determine whether the model should be given more resources, or if the training should be discontinued and new hyperparameter configurations should be evaluated instead. This allows for a strong anytime performance.

Modelling learning curves (MLC) is one multi-fidelity technique that models the learning curve of a configuration, to estimate whether the training of the candidate configuration should be terminated early. There exist different ways of modelling learning curves (Kohavi and John, 1995; Provost et al., 1999), but the general idea is that if it is not predicted that the current model will perform better than the so-far best-performing model, the training of the current model is terminated, and a new configuration of hyperparameters to evaluate is selected. Modelling learning curves might not be that efficient in itself, but when combined with Bayesian optimization, it can significantly improve the optimization performance compared to vanilla Bayesian optimization (Elshawi et al., 2019; Feurer and Hutter, 2019). E.g., Domhan et al. (2015) used learning curve modelling to speed up the state-of-the-art optimization of a deep neural network on the CIFAR-10 dataset by a factor of two.

Another set of multi-fidelity algorithms is the *Bandit-based* algorithms, casting the HPO-problem as a *multi-armed bandit* problem (Mahajan and Teneketzis, 2008). Particularly the two methods *Successive Halving* (SH) and *Hyperband* (HB) have shown great results in optimizing the hyperparameters of deep learning models (Elshawi et al., 2019; Feurer and Hutter, 2019).

Successive Halving (Karnin et al., 2013; Jamieson and Talwalkar, 2016) consists of uniformly allocating a budget to a set of random hyperparameter configurations. Then each configuration is trained within the limitations of the budget and the performance is evaluated. The poor performing half is then dropped, and the budget of the well-performing half is doubled. This is repeated until the best performing configuration is left, and a new set of randomly selected hyperparameter configurations to perform Successive Halving on are drawn. Successive Halving is an efficient approach, but the budget allocation is a major challenge, where one for a given budget has to decide between sampling many configurations on a smaller part of the budget, or a few configurations on a larger part of the budget (Elshawi et al., 2019; Feurer and Hutter, 2019; Yang and Shami, 2020).

Hyperband (Li et al., 2017) was proposed as an extension of Successive Halving to solve the challenge of the number of configurations vs budget size. The method calculates a varying number of configurations, and thus also a varying partial budget size, and uses Successive Halving as an inner routine on these configurations to search for the optimal solution. Hyperband has shown great success for deep learning models and has outperformed random search and vanilla Bayesian optimization (Li et al., 2017; Elshawi et al., 2019; Feurer and Hutter, 2019).

However, Hyperband still constructs new configurations at random and does not utilize the information obtained from testing previous configurations. This limits Hyperband's final performance. Falkner et al. (2018a) then proposed a method, BOHB, that combines Hyperband and Bayesian optimization, with a multivariate

adaption of the TPE as the surrogate model, in an attempt to meet all desired qualities stated at the beginning of this section. The surrogate model is repeatedly used to suggest new hyperparameter configurations, which Hyperband then again is run on. BOHB outperformed both Hyperband and Bayesian optimization on several deep learning applications. Particularly, BOHB achieved state-of-the-art performance for a feed-forward neural network on average over 6 OpenML datasets for computer vision on a large continuous and conditional configuration space.

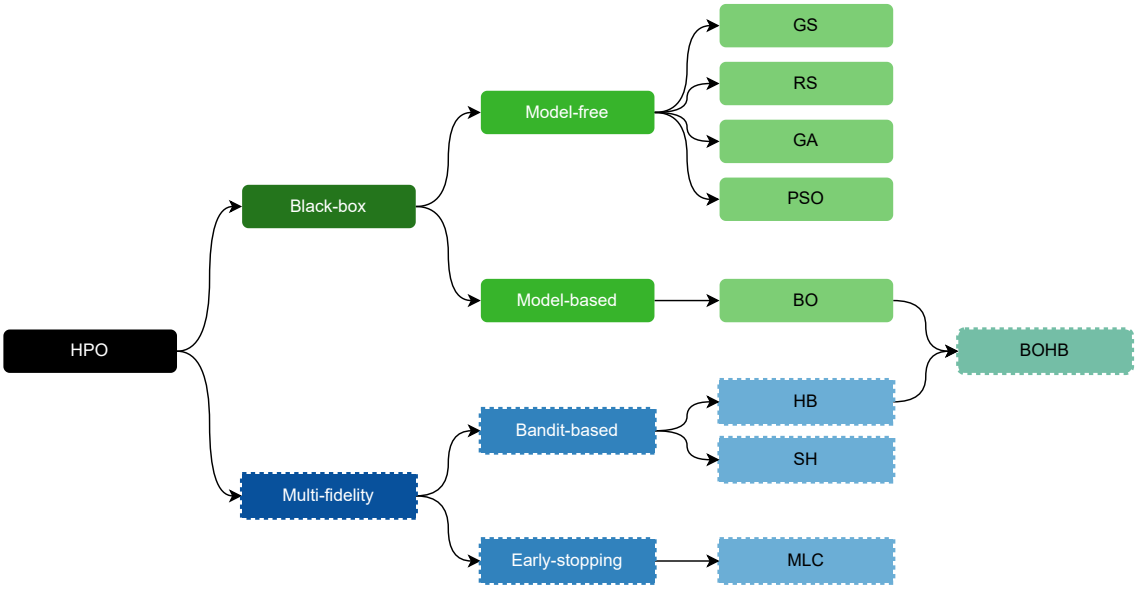


Figure 2.2: Overview of popular HPO-algorithms and multi-fidelity techniques. HPO algorithms are shaded green with a solid line, and MF techniques are shaded blue with a dashed line. BOHB attempts to combine the two approaches. Adapted from Elshawi et al. (2019)

2.2 A note on performance comparison and takeaways

Comparing performances between different HPO-algorithms is a difficult science. It requires extreme amounts of computational power to obtain a statistically robust result, and the optimization algorithms' performances depend highly upon several factors, including, but not limited to:

- Which machine learning algorithm is being used.
- Aspects of the dataset, such as its size and cleanliness.
- The resources available.
- The different domains in the hyperparameter configuration space.

Attempting to benchmark the performance of different HPO-algorithms Klein and Hutter (2019) used four UCI datasets for regression on a purely categorical configuration space and found that BOHB achieved a strong anytime performance, while

BO-TPE achieved a strong final performance. Extending HPO to *Neural Architecture Search*, (NAS) a field closely related to HPO, Ying et al. (2019) experimented with 403k unique architectures trained on the CIFAR-10 dataset. They found that BOHB and SMAC achieved stronger final performance than BO-TPE, while BO-TPE achieved stronger anytime performance than both BOHB and SMAC.

As far as a general recommendation goes, BOHB is recommended as the go-to method for HPO (if multiple fidelities are applicable) by two of the most cited HPO reviews (Feurer and Hutter, 2019; Yang and Shami, 2020) published after BOHB was introduced, and SMAC and BO-TPE are both recommended as alternative methods for HPO of neural networks by the same two reviews if multiple fidelities are not applicable.

Although these recent benchmarks and recommendations indicate some favouring of certain optimization algorithms, there exists no HPO-algorithm that will always perform better than the others. The choice of optimization algorithm has to be tailored to fit your problem at hand.

3 | Theory

This chapter provides the theory needed to understand the case study. First, a gentle introduction to the basic concepts of machine learning are given in Section 3.1. Then (residual) neural networks, the machine learning algorithm of interest in this work, is explained to a greater extent in Section 3.2. Then, finally, hyperparameter optimization is explained in Section 3.3. Both Sections 3.1 and 3.2 are based on Goodfellow et al. (2016).

3.1 Machine Learning

Machine learning, or a machine learning algorithm, is simply an algorithm that automatically is able to discover unknown patterns in data. The data consists of several *examples*, where each example consists of one or more *features*. A neural network, and the model-based optimization methods, are both a form of a *supervised learning* algorithm. These algorithms are algorithms that attempt to learn the output, or the *target*, of corresponding features. For example, given how open a valve is along with the downstream and upstream pressure as features, a supervised learning algorithm could try to learn the target flow rate through the valve. A typical dataset for a supervised machine learning algorithm is illustrated in Table 3.1.

Table 3.1: Dataset example. The table illustrates the differences between examples, features, and targets.

Example #	Feature 1	Feature 2	$\cdots$	Feature m	Target
1	·	·	$\cdots$	·	·
2	·	·	$\cdots$	·	·
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	·	·	$\cdots$	·	·

The process of learning how to recognize patterns is called *training*. When training, a part of the dataset is extracted into a *training set*. The learning algorithm will try to fit the input features of an example in the training set, to the output target of the same example. The difference between the target in the training set, and the model's output, is referred to as the *training error*. After the training is finished, the fitness of the model is evaluated. Evaluating the fitness of a model is done by *testing* the model on a *test set*, which consists of examples not used when training. The testing itself is done by feeding the model features from the test set, and comparing the model's output to the targets in the test set. The difference between the target in the test set, and the model's output, is referred to as the *test error*.

The reason behind splitting the data into a training and test set is to be able to say something about the model's performance on unseen data. This translates to evaluating the model's ability to recognize patterns, rather than just memorize previously seen data. The test error is defined as the expected value of the error on new input. If the model was to be deployed in the real world, where all future data is currently unseen, the test error would tell us what error to expect on that unseen data. What differentiates machine learning from standard optimization, is that by minimizing the training error, we also expect to minimize the test error.

The reason behind the expectation stems from the theory that all data in the dataset is generated by the same *data generating process*, which is a probability distribution over data sets. We assume that all examples in the dataset are independent of each other and that all examples are identically distributed, drawn from the same probability distribution. Thus, the data generating process can be described as a probability distribution over a single example, called the *data generating distribution*, which again allows us to expect that minimizing the training error should also minimize the test error as all data comes from the same probability distribution.

Unfortunately, that is not the case in the real world, and practice shows that the test error is greater than or equal to the training error. The goal of a machine learning algorithm is to obtain a low test error, but since the test error cannot be controlled directly, we are left with two means to achieve that goal:

1. Make the training error small.
2. Make the gap between training and test error small.

The gap between the training and test error is also referred to as the *generalization gap*. From these means arise two central challenges of machine learning:

- *Underfitting*, which occurs when the training error is not small enough, and,
- *Overfitting*, which occurs when the generalization gap is too large.

The patterns that the machine learning algorithm tries to learn, can be looked at as a function from features to target. The model's *capacity* is its ability to fit a wide variety of functions, and by controlling the model's capacity, one can control the model's likelihood to underfit or overfit. Too low capacity might lead to underfitting, while too high capacity might lead to overfitting. A typical relationship between underfitting, overfitting, and capacity is given in Figure 3.1.

As stated earlier, we wish to obtain as low a test error as possible. A simple solution would be to train different models and select the one with the lowest test error. However, this process would lead to *data leakage*, where the data in the test set is no longer unseen, and it therefore cannot be used to say anything about the model's real-world performance. In fact, by selecting the model with the lowest test error we would in reality be minimizing the error of the entire dataset, and not only the training set.

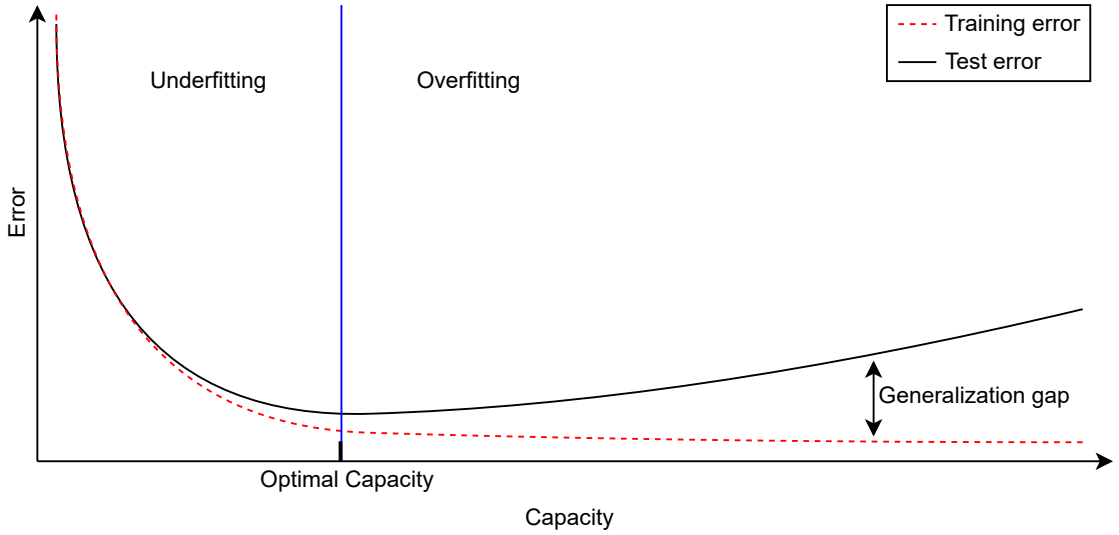


Figure 3.1: Underfitting/Overfitting vs Capacity. A too low capacity might lead to underfitting, while a too high capacity might lead to overfitting. The goal is to find an optimal capacity. Adapted from Goodfellow et al. (2016).

To avoid the problem of data leakage, the dataset is often split into a third part, the *validation set*, which is left unseen during training. *Validation* is performed just like testing, and the *validation error* is calculated the same way as the training and test error. Furthermore, since the validation set is left unseen during training, the validation error is a reasonable estimate of the training error. Rewriting the simple solution from above, after training the multiple models, the model with the lowest validation error is selected. Then that model is further trained on the validation set before the test error is calculated. This leaves the test set unseen, and as such avoid the data leakage problem. This rewritten simple solution is in essence what hyperparameter optimization, and this work, is about.

As stated in the introduction of this section, there exist several machine learning algorithms that train and behave in different ways. This work focuses on a residual neural network, and how these networks are structured, behave and train is discussed in Section 3.2.

3.2 Neural Networks

3.2.1 The structure of a Neural Network

A neural network is built up of *neurons*. A simplified way of looking at a neuron is to consider it as a placeholder for a number (Figure 3.2). This number, 4.3 in the case of Figure 3.2, is referred to as the *activation* of the neuron. To begin with, consider the activation as a simple on/off switch, where an activation less than or equal to zero means off, and an activation greater than zero means on. An activated neuron can be thought of as a neuron that believes it is making the correct prediction, while

a deactivated neuron can be looked at as a neuron that believes that it is not making the correct prediction.

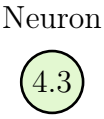


Figure 3.2: Illustration of a neuron with its activation.

These neurons make up the foundation of a neural network and are grouped in *layers*, as illustrated in Figure 3.3. The *input layer* is the first layer of the network and receives the input features from the dataset. We wish to use all features at hand, and thus, the size of the input layer is equal to the number of features in the dataset. The *output layer* is the final layer of the network. The output layer is responsible for driving the model's output towards the targets in the dataset. Therefore, the size of this layer is equal to the number of targets in the dataset. A *hidden layer* is any layer between the input and output layer, and it is called hidden because it is not directly associated with any feature or target in the dataset. The network itself must learn how to use these hidden layers to reach the desired target. The size and structure of these layers are a design choice. The total number of layers makes up the *depth* of the model, and the number of neurons in each layer is the *width* of that layer.

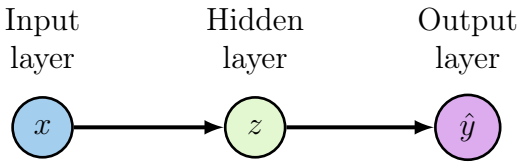


Figure 3.3: Simple illustration of a neural network with one input layer, one hidden layer and one output layer.

Continuing with the flow rate example from Section 3.1, with three features and one target, a possible neural network designed to solve this problem could look like the network illustrated in Figure 3.4, where the size of the hidden layers are chosen purely for illustrative purposes. This network furthermore has two hidden layers and is therefore also called a *Deep Neural Network*. A deep neural network is any neural network that has two or more hidden layers.

The term network implies that there is some sort of connection between multiple neurons. Notice in Figure 3.4 how each neuron in one layer, is connected to each neurone in the next layer. A layer whose neurons are all connected to each neurone in the next layer is referred to as a *fully connected* layer. Furthermore, all connections in the network flow forward as a directed acyclic graph, from the input layer to the output layer. A network with this feature is called a *feed forward network*. The network utilizes these connections to pass information, the different neurons' activation, from one layer to another. As mentioned earlier, the neurons express some sort of belief, or some sort of opinion. And just as humans are able to decide how much to believe in another human's opinion, these neurons also need to be able to decide how much they value the input from another neuron.

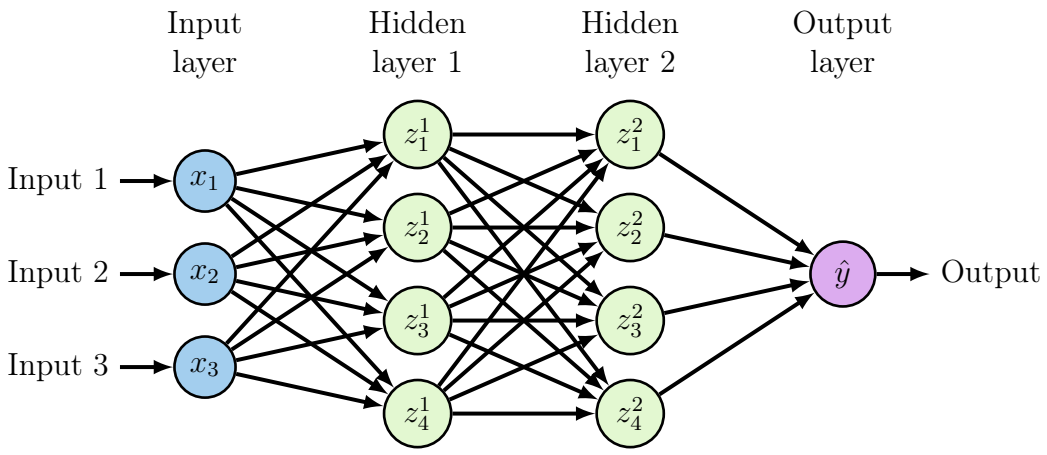


Figure 3.4: Simple illustration of a Deep Feed-Forward Neural Network with one input layer, two hidden layers and one output layer. The network is called deep because it has two or more hidden layers, and it is called feed forward since all connections flow forwards in a directed acyclic graph.

Therefore, a connection between a neuron n_i^{L-1}, in layer $L-1$, and n_j^L, in layer L, has its own associated weight, w_{ij}^L, as illustrated in Figure 3.5. The receiving neuron then calculates the weighted sum of all its inputs, to reach its own opinion, or activation:

$$z_j^L = \sum_{i=1}^{I} z_i^{L-1} * w_{ij}^L,$$

where z_j^L is the activation neuron j in layer L, z_i^{L-1} is the activation neuron i in layer $L-1$, w_{ij}^L is the weighted connection between the corresponding neurons, and I is the number of neurons in layer $L-1$.

In addition to the weighted sum, a *bias* is added to add some sense of meaningfulness to the activation, i.e., to be able to say that e.g., the activation is only meaningful if the weighted sum is above a certain threshold. Adding this bias yields the following activation:

$$z_j^L = b_j^L + \sum_{i=1}^{I} z_i^{L-1} * w_{ij}^L, \tag{3.1}$$

where again z_j^L is the activation neuron j in layer L, z_i^{L-1} is the activation neuron i in layer $L-1$, w_{ij}^L is the weighted connection between the corresponding neurons, b_j^L is the bias of neuron j in layer L, and I is the number of neurons in layer $L-1$.

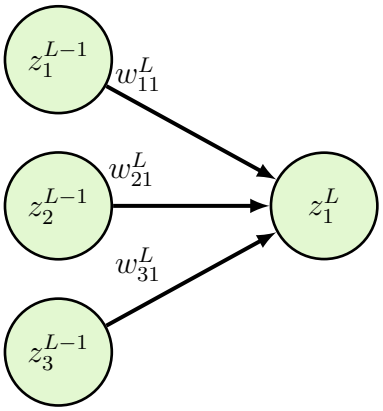


Figure 3.5: Weighted connections from layer $L-1$ to layer L, with arbitrary activation values z_i^l.

The weighted sum of Equation 3.1 is a linear transformation, and thus we can express the activation of an entire layer as a matrix-vector equation:

$$\underbrace{\begin{bmatrix} z_0^L \\ z_1^L \\ \vdots \\ z_j^L \end{bmatrix}}_{\boldsymbol{z}^L} = \underbrace{\begin{bmatrix} w_{00}^L & w_{01}^L & \cdots & w_{0j}^L \\ w_{10}^L & w_{11}^L & \cdots & w_{1j}^L \\ \vdots & \vdots & \ddots & \vdots \\ w_{i0}^L & w_{01}^L & \cdots & w_{ij}^L \end{bmatrix}}_{\boldsymbol{W}^L} \underbrace{\begin{bmatrix} z_0^{L-1} \\ z_1^{L-1} \\ \vdots \\ z_j^{L-1} \end{bmatrix}}_{\boldsymbol{z}^{L-1}} + \underbrace{\begin{bmatrix} b_0^L \\ b_1^L \\ \vdots \\ z_i^L \end{bmatrix}}_{\boldsymbol{b}^L} \tag{3.2}$$

From Equation 3.2 it is clear to see that the activation of the output layer is just a linear transformation of the input, illustrated below according to Figure 3.4

$$\begin{aligned} \boldsymbol{z}^1 &= \boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1 \\ \boldsymbol{z}^2 &= \boldsymbol{W}^2\boldsymbol{z}^1 + \boldsymbol{b}^2 = \boldsymbol{W}^2(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2 \\ \hat{y} &= \boldsymbol{W}^y\boldsymbol{z}^2 + \boldsymbol{b}^y = \boldsymbol{W}^y\Big(\boldsymbol{W}^2(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2\Big) + \boldsymbol{b}^y \end{aligned} \tag{3.3}$$

Although this linear transformation might be able to recognize some patterns, it is not able to express any nonlinearity. Since the goal of a neural network is to be able to predict values of real-world processes, which are mostly nonlinear, some nonlinearity has to be introduced to Equation 3.3. We introduce the nonlinearity by adding a nonlinear *activation function*, a, that wraps the weighted sum of Equation 3.1 as such:

$$z_j^L = a\Big(b_j^L + \sum_{i=1}^{I} z_i^{L-1} * w_{ij}^L\Big) \tag{3.4}$$

The output of the neural network can thus be written as a composite mapping of the activation function in the different layers from the input $\boldsymbol{x}$:

$$\begin{aligned}\hat{y} &= (a^y \circ a^2 \circ a^1)(\boldsymbol{x}) \\ &= a^y\Big(\boldsymbol{W}^y a^2\big(\boldsymbol{W}^2 a^1(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2\big) + \boldsymbol{b}^y\Big)\end{aligned} \tag{3.5}$$

where a^l is the activation function in layer l.

3.2.2 Training of a Neural Network

Simply put, a neural network trains by tweaking its weights and biases, or *learning parameters*, until it reaches a satisfying point. It does so by using a forward-backwards propagation scheme. *Forward* propagation is what is described in Equation 3.5, where an input is given to the network, and the activations of the neurons are propagated through the layers until the output layer makes the final estimation and a training loss can be calculated.

As stated in Section 3.1, the goal of the training process is to minimize the training loss, such that the test loss also will be minimized. To perform any minimization at all, a metric to measure the training loss must be defined. This metric is obtained with the use of a *cost function*, $J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$, that measures the difference between the expected outputs, $\boldsymbol{y}$, and the network's outputs to the corresponding inputs $\boldsymbol{x}$, for a set of learning parameters θ, and yields a scalar cost.

The cost represents how wrong the networks' prediction was, or in other words, how wrong the learning parameters were. *Backward* propagation then refers to the method of how these weights and biases are updated based on the response of the cost function. In particular, a neural network propagates the cost backwards (Figure 3.6), from the output layer to the input layer, and uses *gradient descent* to update the learning parameters to minimize the cost function:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k; \boldsymbol{x}, \boldsymbol{y}), \tag{3.6}$$

where the learning parameters $\theta = (\boldsymbol{w}_i^L, b_i^L)$, α is the step length in the gradient descent method, or in ML-terms the *learning rate*, $\nabla_\theta J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$ is the gradient of J with respect to $\boldsymbol{\theta}$, and k in subscript refers to an arbitrary iteration. In machine learning terms, the gradient descent method is referred to as the *optimizer*.

Since J is some function of y, which by generalizing Equation 3.5 to

$$\boldsymbol{y} = (a^L \circ a^{L-1} \circ a^{L-2} \circ \cdots \circ a^0)(\boldsymbol{x}) \tag{3.7}$$

is a function of all activations of weights and biases in the network, it is clear to see that the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$ is also dependent on all activations of weights and biases in the network. The gradient, therefore, tells us how the change of a single weight or bias would influence the cost. This information is then again used to update the learning parameters of the network to step closer to a minimum of

the cost function. Exactly how the gradient is calculated can be seen in Goodfellow et al. (2016, Chapter 6.5).

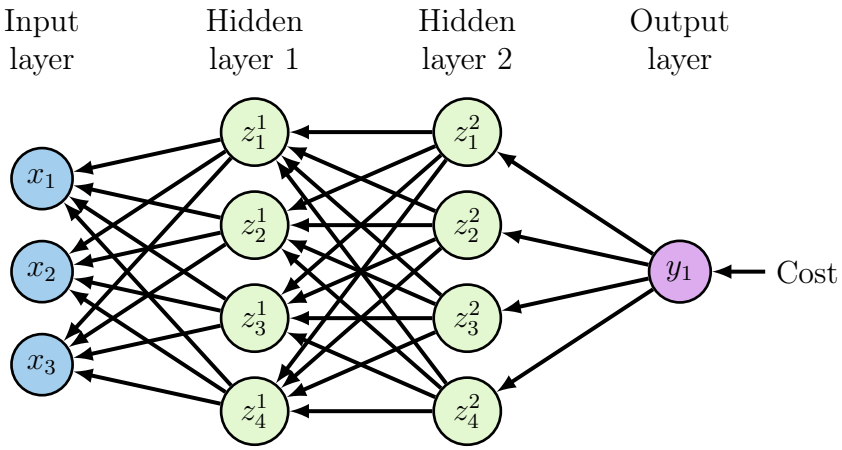


Figure 3.6: Illustration of backwards propagation for the network depicted in Figure 3.4.

Equation 3.7 explains how the output is obtained for a single training example. Usually, with neural networks, a dataset would have, at least, tens of thousands of training examples. A single training iteration, or an *epoch* in ML terms, refers to when the entire training set has been propagated forwards, then backwards and the weights and biases have been updated. For a dataset with n training examples, the *batch gradient descent* (BGD), would for a single epoch require calculating the average gradient of all training examples before taking a single step:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\alpha}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}). \tag{3.8}$$

Although the step would be quite a good step, it is too computationally expensive for large datasets, as the entire training set would have to be loaded into memory. *Stochastic Gradient Descent* (SGD), however, uses a single training example to approximate the gradient:

$$\frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) \approx \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}). \tag{3.9}$$

By selecting examples from the dataset uniformly at random, the SGD uses an unbiased estimation of the full gradient:

$$\mathbb{E}_{i \sim \mathcal{U}(1,n)}[\nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})] = \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$$

For a single epoch, SGD results in more gradient descent steps, as a single step is taken for every example in the training set. Each step taken by SGD is also noisier than with batch gradient descent, as SGD uses an approximation of the gradient. Since more steps are taken, the computational time of SGD is also higher than for batch gradient descent. It is in turn however less computationally expensive to use SGD, as only one training example has to be loaded into memory at once.

Mini-batch gradient descent (MBGD) is a method that aims to compromise between SGD and BGD. Instead of using either all the training examples or a single example, MBGD uses a subset of the examples, a *mini-batch*, $j \in (1, \ldots, n)$, to approximate the gradient:

$$\frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) \approx \nabla_{\boldsymbol{\theta}} J_j(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}). \tag{3.10}$$

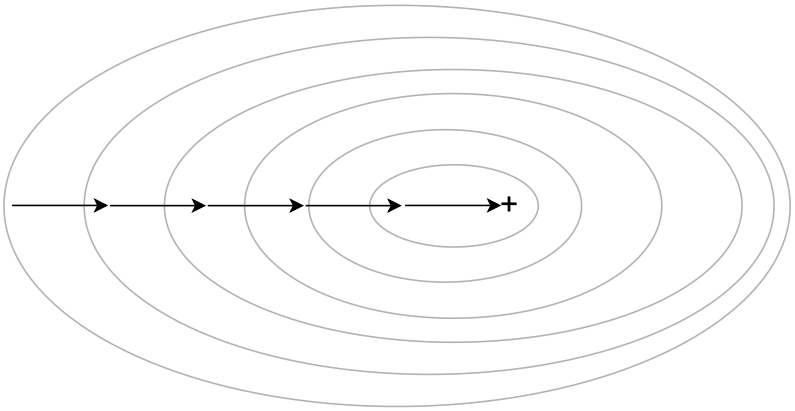
Which again is an unbiased estimate of the gradient:

$$\mathbb{E}_{j\sim\mathcal{U}(1,n)}[\nabla_{\boldsymbol{\theta}} J_j(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})] = \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$$

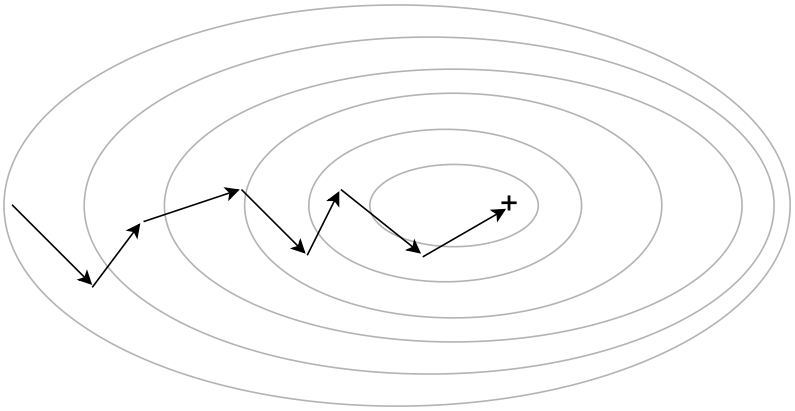
An illustrative comparison between SGD, BGD and MBGD is given in Figure 3.7.

Taking a step back to look at the cost function itself, it is as stated a function of the output of the neural network, which again is a composite function of nonlinear functions. This mapping results in the cost function being a non-convex functional, which means that the gradient descent might not always converge towards the global minimum. However, '*experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost*' (Goodfellow et al., 2016). This idea is illustrated in Figure 3.8.

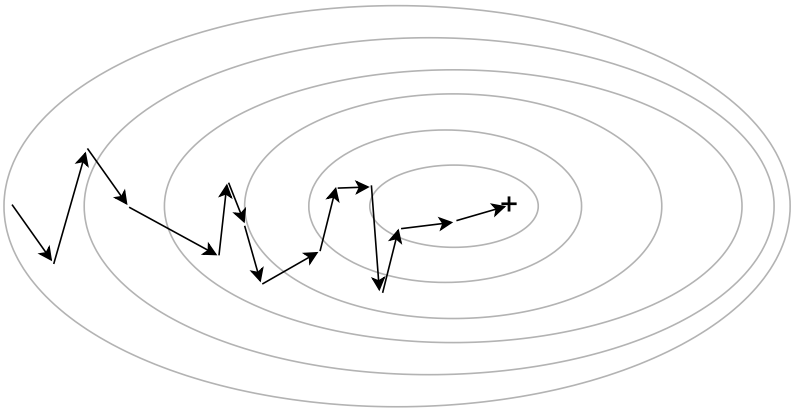
An important part of the gradient descent method not yet discussed is the learning rate α of Equation 3.6. One might consider the gradient to be a linear approximation to the non-linear non-convex cost function. For a relatively non-linear cost function, the approximation might be good in some area around where the gradient is evaluated. If the cost function is highly non-linear, on the other hand, the area where the approximation is good decreases. The learning rate controls the size of the step to make sure that the learning parameters aren't moved too much out of that good area. A too high learning rate can result in too rapid learning, as the network might not be able to effectively gradually decrease the loss, and it might overshoot a good local minimum. A too low learning rate can result in too slow learning, where the network uses too much time to converge on a good local minimum, converges on a poor local minimum, or might not converge at all. Essentially, a too high learning rate believes too much in the linear approximation, while a too low learning rate believes too little in the linear approximation. A figurative illustration of the effect the learning rate has is given in Figure 3.9.



(a) Batch Gradient Descent



(b) Mini-batch Gradient Descent



(c) Stochastic Gradient Descent

Figure 3.7: Illustration of different gradient descent variants: BGD **(a)** uses fewer steps with low variance, while SGD **(c)** uses more steps with high variance. MBGD **(b)** is a compromise between the other two.

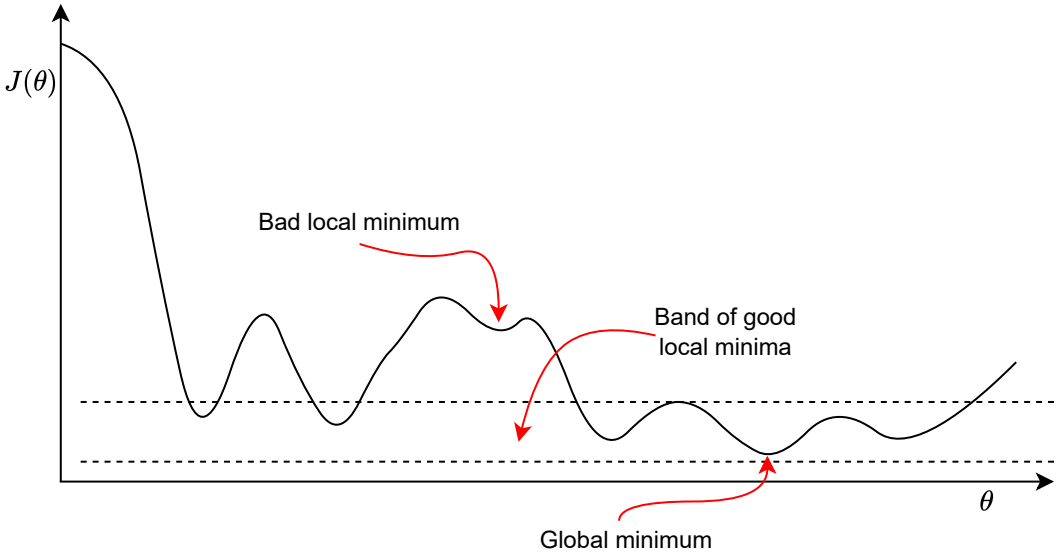


Figure 3.8: Most local minima of a neural network's cost function are good. The goal of minimizing is to arrive at one of these good local minima, instead of necessarily finding the global minimum. Adapted from Goodfellow et al. (2016).

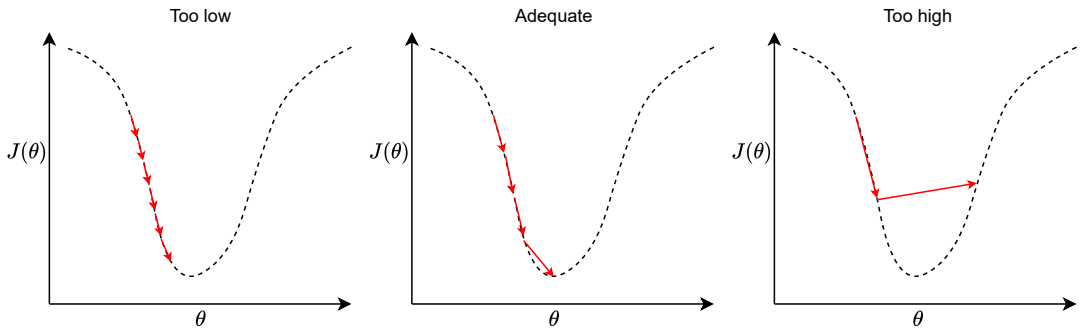


Figure 3.9: Too low a learning rate (left) can lead to slow convergence or no convergence at all. Too high a learning rate (right) can skip a good local minimum. An adequate learning rate (centre) is somewhere in between.Adapted from Goodfellow et al. (2016).

3.2.3 Weight initialization

One interesting aspect of using GD methods is that they require an initial starting point, i.e., an initial set of weights and biases. The starting point can highly affect the training process, and might even more or less single-handedly determine the final performance of the network. Thus, properly initializing the learning parameters is important to achieve satisfying results.

Not much is yet understood about how exactly the parameter initialization affects the performance of the model. Thus, most initialization techniques are simple and 'based on achieving some nice properties when the network is initialized' (Goodfellow et al., 2016).

One property that is known with certainty to be important is to break the symmetry

between different units; '*If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.*' (Goodfellow et al., 2016).

There exist different ways of breaking the symmetry, but a computationally effective, simple and the most used method is to draw initial parameters at random from a Gaussian or uniform distribution. To simplify the process, the biases are often initialized to a constant value, such that only the initial weights are randomly selected, and hence the name *weight initialization*. The constant value is furthermore often set to zero, so as to not propose any assumptions on the network.

The mean of the Gaussian or uniform distribution is also for the same reason often set to zero. The variance, or bounds, however, varies based on which initialization technique is used. The choice of technique often depends upon which activation function is being used. That is mainly because different activation functions have different derivative behaviour. Kumar (2017) provides a great overview of initialization techniques for different activation functions. For more on weight initialization, please see Goodfellow et al. (2016, Section 8.4).

3.2.4 Regularization

The goal as stated in Section 3.1 was to minimize the training loss, in order to also minimize the test loss. Section 3.2.2 discussed how the training loss is minimized. *Regularization* are techniques used to reduce the test error by other means than directly reducing the training error. In other words, regularization tries to move the model's capacity from the overfitting regime, toward the optimal capacity.

There exists several ways of regularizing a deep neural network (Goodfellow et al., 2016, Chapter 7), but the regularization technique used in this work is *parameter norm penalties*. Parameter norm penalties works by adding a norm penalty, $\Omega(\boldsymbol{\theta})$, to the cost function:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) + \kappa\Omega(\boldsymbol{\theta}), \tag{3.11}$$

where $\tilde{J}$ is the regularized cost function, κ is a parameter that weights the penalty function. A high κ will result in high regularization, while a smaller κ results in lesser regularization.

When the regularized cost function is minimized, it will along with the original cost function J also decrease some measure of the size of the learning parameters, $\boldsymbol{\theta}$. Particularly, what the norm penalty does is that it is penalizing the large weights of the neural network. This is penalization is used to avoid some weights dominating the gradient, and thus influencing the step direction superiorly. Regularizing a network often leads to a network with lower test error than the same network without regularization would achieve.

3.2.5 Residual blocks

The *universal approximation theorem* states that a feed-forward network of a single hidden layer with a suitable activation function and a linear output layer is sufficient to represent any continuous function on a compact domain to any desired accuracy (Goodfellow et al., 2016; Lu et al., 2017). Consider an extreme example where every possible input feature is known. A wide enough network would then be able to memorize every possible corresponding target. This is not useful for any practical application, as the true data generation process is not known, and practice shows that these networks tend to overfit.

The idea of using multiple layers is that each layer can learn different abstractions of the pattern from feature to target. Increasing the number of layers often reduces the number of neurons required to represent the desired pattern, and can reduce the test error. However, simply stacking more and more layers on top of each other leads to the *degradation problem*, where the introduction of more hidden layers saturates the training loss before it eventually starts increasing. Thus, an increase in test loss is not due to overfitting, but to the training going in the wrong direction.

The *residual block* Figure 3.10 was introduced by He et al. (2016) to address the degradation problem. Consider a shallower net and a deeper net that consists of the shallower net plus some more added layers, where the added layers are identity mapping ($f(x) = x$). The deeper net should then produce no higher training loss than the shallower net. However, that is not the case in practice.

Let $\mathcal{H}(\boldsymbol{x})$ denote the original desired underlying mapping. Instead of adding layers to fit $\mathcal{H}(\boldsymbol{x})$, the authors rather proposed to let the added layers approximate the residual $\mathcal{F}(\boldsymbol{x}) = \mathcal{H}(\boldsymbol{x}) - \boldsymbol{x}$, which recasts the original mapping to $\mathcal{H}(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$. The latter equation is realized by the use of *skip connections*, that allows information to jump over layers, as illustrated in Figure 3.10.

With this reformulation, and assuming that the identity mappings are optimal, the problem of finding the optimal network simply reduces to driving the weights of the layers in the residual block to zero. In practice, it is unlikely that the identity mappings are optimal, but the authors showed that it might reasonable to assume that the optimal is closer to an identity mapping than to a zero mapping. It should thus be easier to learn with the residual mapping $\mathcal{F}(\boldsymbol{x})$, rather than the original desired underlying mapping $\mathcal{H}(\boldsymbol{x})$.

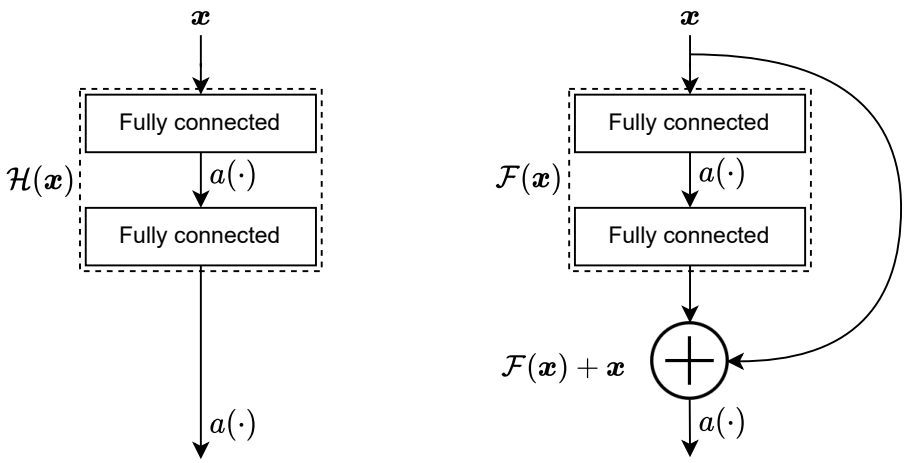


Figure 3.10: Example of a regular block (left) and a residual block (right). The curved edge from the input to the addition signifies a skip connection. $g(\cdot)$ is the activation function. Note that $\mathcal{H}(\boldsymbol{x})$, and thus also $\mathcal{F}(\boldsymbol{x})$, can have many different designs. Adapted from He et al. (2016)

3.3 Hyperparameter optimization

Section 3.2.2 introduced how the training error is minimized, and Section 3.2.4 described a technique to alter the training process to make the generalization gap smaller. The goal of this work is to control the training process such that we achieve both a low training loss and a small generalization gap. This work does so by optimizing the *hyperparameters* of the model.

A hyperparameter is a parameter that is used to control the training of a machine learning model. These are parameters that are decided beforehand, and opposite to the learning parameters, are not updated throughout the training. The depth of the network, the width of a layer, the learning rate, the activation function, the mini-batch size, and the norm penalty are all examples of different hyperparameters.

The definition of a hyperparameter is quite loose, and there are several other "parameters" that also contribute to the behaviour of the learning process. A parameter could, e.g., be whether to use residual blocks or not in the architectural design of the network, which certainly would affect the training. However, such an architectural design parameter is not a hyperparameter but is instead a part of another interesting machine learning topic called neural architecture search (Elsken et al., 2019), which aims to find the best-suited model architecture.

Stretching it even further, by looking at the dataset, which certainly also affects the training, one could define parameters for this as well. E.g., removing examples with features that lie outside some factor of the interquartile range for that feature, or selecting to use only a subset of all features. By incorporating this *data cleaning* and *feature selection*, respectively, we are approaching another interesting topic called *automated machine learning* (Auto ML) (Hutter, Kotthoff et al., 2019), which aims to automate the entire machine learning process.

With hyperparameter optimization, the goal is to extract the optimal out of a selected architecture, where data has been preprocessed and features are already selected. Figure 3.11 illustrates this relationship.

Figure 3.11: HPO in the AutoML chain. HPO aims to extract the maximum out of a model that already exists.

3.3.1 Problem statement

Rewriting the problem definition from Feurer and Hutter (2019):

Definition 3.1 (Hyperparameter optimization)**:**
Let $\mathcal{A}$ denote a machine learning algorithm with N hyperparameters.

We denote the domain of the n-th hyperparameter by Λ_n, and the overall hyperparameter configuration space as $\boldsymbol{\Lambda} = \Lambda_1 \times \Lambda_2 \times \ldots \Lambda_N$.

A vector of hyperparameters is denoted by $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$, and $\mathcal{A}$ with its hyperparameters instantiated to $\boldsymbol{\lambda}$ is denoted by $\mathcal{A}_{\boldsymbol{\lambda}}$.

Given a dataset $\mathcal{D}$, our goal is to find

$\boldsymbol{\lambda}^* = \text{argmin}_{\boldsymbol{\lambda} \in \Lambda} \, \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathcal{D}}[V(\mathcal{A}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})]$,

where $V(\mathcal{A}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})$ validates the loss of a model generated by algorithm $\mathcal{A}$ with hyperparameters $\boldsymbol{\lambda}$ on training set D_{train} and evaluated on validation set D_{valid}.

The objective is thus a black-box functional from hyperparameters $\boldsymbol{\lambda}$ to validation loss $V(\mathcal{A}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})$, where no convexity can be assumed on the response surface. For simplicity, we will shorten the objective as $f(\boldsymbol{\lambda})$, and denote the corresponding validation loss as v, such that $v = f(\boldsymbol{\lambda})$.

The only way to obtain accurate information about the objective is to sample the response. By sampling, we mean training the neural network with its hyperparameters instantiated to some value, and calculating the validation loss. Appropriately limiting the size of the configuration space is therefore paramount to achieving satisfying results with any HPO algorithm. Consider, e.g., a configuration space of five hyperparameters, where each hyperparameter can take on a modest 10 unique values. The configuration space thus have a total of $100,000$ possible configurations. If it takes 30 minutes to train the model, it would take approximately 6 years to test every configuration, if they were to be run sequentially on one machine.

Furthermore, the unique values hyperparameters can take on may come from different domains. For example, the learning rate can be real-valued, the depth can

be integer-valued, and which activation function to use can be categorical. It can also exist conditional relationships between hyperparameters, such as which width a layer should have, if the layer exists. Therefore, the unique values a hyperparameter can take on are often much greater than 10, highlighting again the importance of limiting the configuration space.

Another technique used to make hyperparameter optimization more feasible is to make informative decisions based on information from previous trials, such as Bayesian optimization does.

3.3.2 Bayesian Optimization

Bayesian optimization (Brochu et al., 2010; Shahriari et al., 2015) is a sequential model-based optimization (Hutter, Hoos et al., 2011) algorithm designed for global, expensive black-box optimization. An SMBO algorithm consists of two main components:

1. A surrogate, M, that models the response surface of the objective.
2. An acquisition function, u, that based on the surrogate decides where to sample next.

The idea behind SMBO is that by minimizing the surrogate, one also minimizes the true objective. SMBO does so by iteratively sampling and building up the surrogate. The acquisition function acts as a broker that decides where it is clever to sample next. Pseudocode for generic SMBO is given in Algorithm 1.

Algorithm 1 Pseudocode for Sequential Model Based Optimization. Adapted from Bergstra, Bardenet et al. (2011)

Require: M_0, T, a, f
$\mathcal{H} \leftarrow \emptyset$, $M_0 \leftarrow \emptyset$
for $t \leftarrow 1$ **to** T **do**
  $\boldsymbol{\lambda}' \leftarrow \text{argmin}_{\lambda}\, u(\boldsymbol{\lambda}, M_{t-1})$ ▷ Acquisition function
  Train $\mathcal{A}_{\boldsymbol{\lambda}'}$ and evaluate $v = f(\boldsymbol{\lambda}')$
  $\mathcal{H} \leftarrow \mathcal{H} \cup (\boldsymbol{\lambda}', v)$
  Fit a new model M_t to $\mathcal{H}$ ▷ Surrogate model
end for
return $\mathcal{H}$

The acquisition function used by the selected optimization algorithms in this work is the *Expected Improvement* (EI) criteria, which maximizes how much improvement one can expect to achieve from some threshold v^*:

$$\text{EI}_{v^*}(\lambda) := \int_{-\infty}^{\infty} \max\left(v^* - v, 0\right) p_M(v \mid \lambda) dy \tag{3.12}$$

Since sampling the objective is expensive, constructing a good surrogate model within relatively few samples is important. Bayesian optimization does so in a particularly effective fashion. As the name suggests, it builds the surrogate as a posterior distribution using Bayes' rule:

$$\begin{aligned} M = p(v|\lambda) \quad &\propto p(v) \ \times p(\lambda \mid v) \\ M = \text{posterior} \quad &\propto \text{prior} \times \text{likelihood} \end{aligned}$$

Whereas e.g., Bayesian optimization with Gaussian Processes (Williams and Rasmussen, 2006) models the posterior directly, the selected HPO algorithms of this work use the tree-structured Parzen estimator, which instead models the likelihood.

Tree-structured Parzen Estimator

The tree-structured Parzen estimator was proposed by Bergstra, Bardenet et al. (2011), and this section is mainly inspired by their work.

Let us first consider the tree part. The authors consider the HPO problem as a problem of 'optimizing a loss function over a graph-structured configuration space', but restrict themselves to a tree-structured graph. The configuration space is tree-structured, in 'the sense that some leaf variables are only well-defined when node variables take particular values.' E.g., which width a layer should have (leaf), if the layer exists (node).

The authors further define a configuration space by a 'generative process for drawing valid samples'. That is, first choose if the layer should exist, then choose the width of that layer. The TPE then models the likelihood $p(\lambda \mid v)$ by 'transforming that generative process, replacing the distributions of the configuration prior with non-parametric densities.' The densities are obtained by using kernel density estimation (KDE), or Parzen-window estimation, which is where the TPE gets its Parzen estimator part from.

Kernel density estimation is a non-parametric way of estimating the probability density function from observed data. The TPE does so by stacking Gaussian kernels (or Gaussian distributions) with mean corresponding to the value of the observed hyperparameter, and standard deviation equal to the maximum distance to the left and right neighbour of the observation, limited to some threshold.

The TPE splits the samples into two categories. One category, ℓ, where $v = f(\lambda)$ is less than some threshold v^*, and one category, g, containing all the other samples. It then models $p(\lambda \mid v)$ by modelling one density for each category:

$$p(\lambda \mid v) = \begin{cases} \ell(\lambda) & \text{if } v < v^* \\ g(\lambda) & \text{if } v \geq v^* \end{cases} \tag{3.13}$$

v^* is chosen such that $p(v < v^*) = \gamma$, where, γ is some quantile of the observed samples v. The authors further show that by this construction, the EI criteria can be written as

$$\begin{aligned} \mathrm{EI}_{v^*}(\lambda) &= \int_{-\infty}^{v^*} (v^* - v)\, p(v \mid \lambda) dv \\ &= \int_{-\infty}^{v^*} (v^* - v)\, \frac{p(\lambda \mid v) p(v)}{p(\lambda)} dv \\ &\propto \left(\gamma + \frac{g(\lambda)}{\ell(\lambda)}(1-\gamma)\right)^{-1} \end{aligned} \tag{3.14}$$

Equation 3.14 shows that the expected improvement tries to maximize the probability of a hyperparameter being in ℓ, while minimizing the probability of the same hyperparameter being in g. BO-TPE utilizes this by first drawing many samples from ℓ and then evaluating them according $g(\lambda)/\ell(\lambda)$. Equation 3.14 also shows that no specific model for $p(v)$ is needed. An illustration of the relationship from samples, to categories, to density and utility, is given in Figure 3.12.

An important aspect of the TPE is that it uses multiple 1-dimensional estimators. I.e., it models only the probability density function of a single hyperparameter being in ℓ or g, and does not model any covariance with any other hyperparameter. So, with the generative process used to select the next candidate of hyperparameters to sample, it would first decide an appropriate value for λ_1, and then decide a value for λ_2 completely disregarding the value of λ_1 (if there is not a conditional relationship between λ_1 and λ_2).

Whether the use of multiple univariate density estimators is an advantage or disadvantage is unclear. E.g., Klein and Hutter (2019) attributes the superior performance of BO-TPE over similar methods to the use of univariate KDEs, while Ying et al. (2019) attributes the inferior performance of the BO-TPE to the same reason.

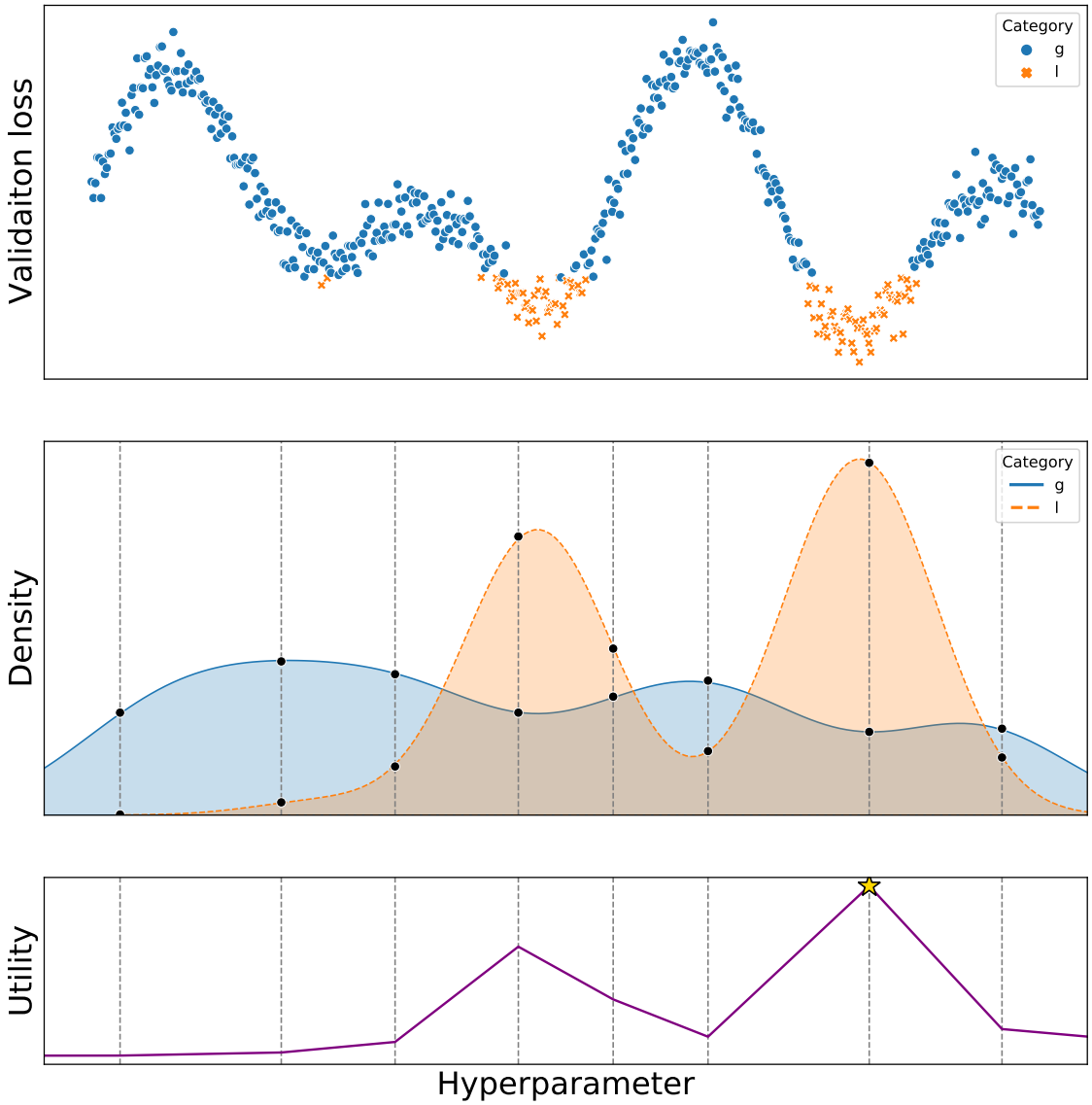


Figure 3.12: BO-TPE illustration. Top: Samples are split into two categories. Middle: KDE of data from the two categories. Black dots illustrate samples. Bottom: Expected improvement at the sampled points. The star indicates the highest expected improvement. The next hyperparameter configuration will use the hyperparameter value of that point.

3.3.3 Multi-Fidelity Optimization

The BO-TPE algorithm (and most other vanilla SMBO algorithms) only evaluates a model's performance after the net has reached some convergence criterion. For a given model there could be several, simultaneous, criteria, such as e.g., the training or validation loss not decreasing for some epochs, a fixed number of epochs have passed or an amount of wall-clock time has passed.

With multi-fidelity optimization, the question asked is: 'Can we identify and terminate poor-performing hyperparameter settings early in a principled online fashion to speed up hyperparameter optimization?' (Jamieson and Talwalkar, 2016)

Early stopping

Early stopping is one technique that tries to answer the multi-fidelity question and involves stopping the training process before the model reaches one of its convergence criteria. In this work, we will use a predefined fixed number of epochs as the convergence criteria, also referred to as the *budget*.

Given a budget, one must decide some way if the model should be given the full budget, or if it should be early stopped. This decision is referred to as a *stopping rule*. One simple, yet effective, non-parametric stopping rule is the *median stopping rule* (Golovin et al., 2017, Section 3.2.2). The median stopping rule (MSR) stops a trial, if the trial's best-observed validation loss is strictly less than the median of the running average of previous trials up until similar points in time. By using the median to compare, the MSR is fairly robust to outliers, as a horribly bad configuration would not affect the median any more than a moderately bad configuration would. See Figure 3.13 for illustration. Combining the median stopping rule with SMBO is trivial.

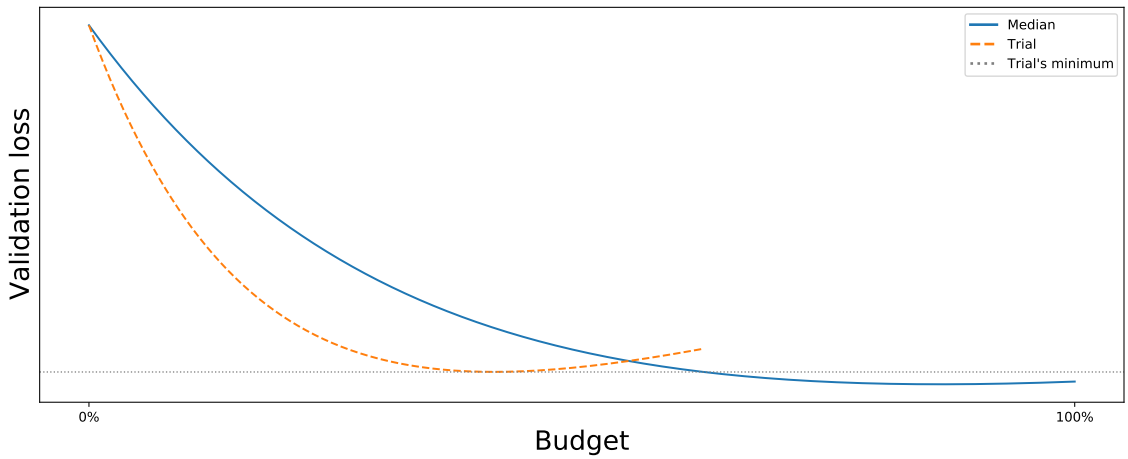


Figure 3.13: Median Stopping Rule. The trial is stopped after its best-observed validation loss is strictly less than the median up until similar points in time. This can lead to allowing the trial to continue, even though its loss is higher than the median, as the figure illustrates.

Bandit based approaches

Bandit based hyperparameter optimization approaches try to answer the multi-fidelity question by casting the HPO problem as an instance of 'non-stochastic best-arm identification' (Jamieson and Talwalkar, 2016). The general best arm problem for multi-armed bandits is given in Algorithm 2.

In the context of hyperparameter optimization, each arm corresponds to a fixed hyperparameter configuration, pulling an arm corresponds to a fixed number of epochs, and the loss corresponds to the validation loss. At each step, the algorithm only observes hyperparameter configuration I_t, and the goal is to choose a hyperparameter configuration J_t such that J_t has the lowest validation loss.

Algorithm 2 Best Arm Problem for Multi-armed Bandits. $l_{i,k}$ denotes the loss observed on the kth pull of the ith arm. Adapted from Jamieson and Talwalkar (2016)

Require: n arms
$T_i \leftarrow 1 \quad \forall \quad i \in [n]$
$t \leftarrow 1$
repeat
    Algorithm chooses an index $I_t \in [n]$
    Loss $l_{I_t,T_{I_t}}$ is revealed
    $T_{I_t} \leftarrow T_{I_t} + 1$
    Algorithm outputs a recommendation $J_t \in [n]$
    $t \leftarrow t + 1$
until External stop signal received

Successive Halving

To answer the multi-fidelity question, Jamieson and Talwalkar (2016) proposed a solution using the Successive Halving method from Karnin et al. (2013): Uniformly allocate a given budget to a set of arms for a predefined number of epochs. Then, evaluate the arms' performance, stop the worst performance half, double the budget of the good performing half, and repeat until just one arm is left. An illustration of Successive halving algorithm is given in Figure 3.14. Pseudocode for the algorithm is given in Algorithm 3

Algorithm 3 Successive Halving. Adapted from Jamieson and Talwalkar (2016)

Require: Budget B, n number of configurations
$s_0 \leftarrow$ getHyperparameterConfigurations(n)
$s_{max} \leftarrow \lceil \log_2(n) \rceil$
for $k = 0, 1, \ldots, s_{max} - 1$ **do**
    $r_k \leftarrow \lfloor \frac{B}{|s_k| s_{max}} \rfloor$
    $L \leftarrow \text{sample}(s, r_k) \quad \forall s \in s_k$
    $s_{k+1} \leftarrow \text{topHalf}(s_k, L)$
end for
return Configuration with the smallest intermediate loss seen so far.

- getHyperparameterConfigurations(n) - Returns n hyperparameter configurations drawn random uniformly from the hyperparameter configuration space.
- sample(s, r) - Trains the model with hyperparameter configuration t for r epochs and returns the validation loss observed after r epochs.
- topHalf(s, L) - Returns the top half performing configurations from s, based on the corresponding losses L.

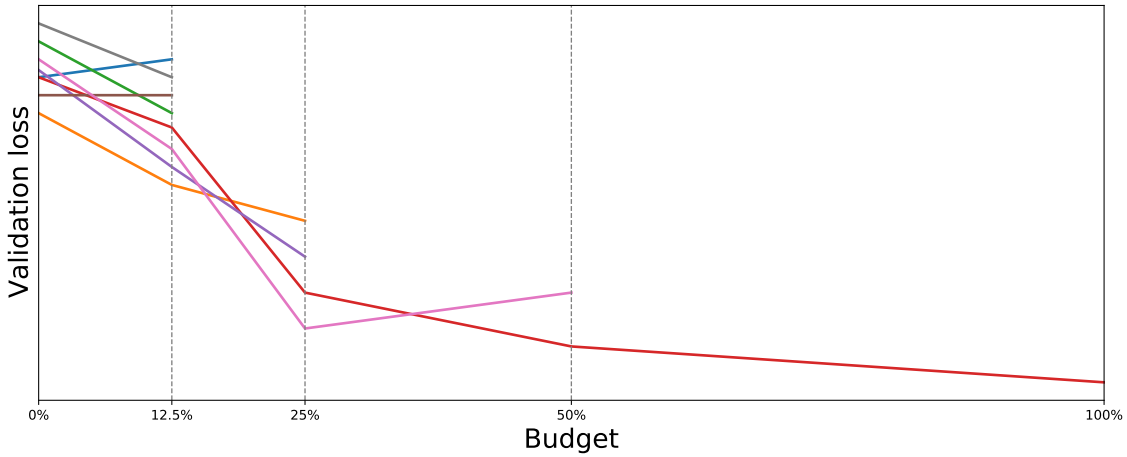


Figure 3.14: Successive Halving, illustrated with 8 configurations. Budgets are doubled when half of the configurations are stopped. Only one configuration is allowed to use the full budget.

There is one major challenge of the Successive halving algorithm, namely the size of the budget vs the number of configurations. For a given budget B, Successive Halving allocates on average B/n resources to the configurations. Thus, the algorithm requires choosing whether a few configurations with a larger budget on average or many configurations with a smaller budget on average should be considered. There is no general solution to this trade-off.

Hyperband

Hyperband (Li et al., 2017) addresses the trade-off problem by considering multiple numbers of arms n for a given budget B. Pseudocode for the Hyperband algorithm is given in Algorithm 4. The input R is the maximum resources a single configuration can be given, and η determines the proportion of trials discarded in each iteration of Successive Halving. The default value of η is 3, which means that two-thirds of the trials are discarded in every iteration of Successive Halving.

The algorithm consists of an outer loop (lines 4-6) and an inner loop (lines 8-11). The outer loop calculates different values of the number of configurations to sample, n, and the minimum number of resources, r, allocated to each configuration. Each run of the outer loop is called a *bracket*.

Each bracket invokes a single run of Successive Halving with fixed values n and r, and uses approximately B total resources. The brackets are run in an order that favours exploring first, i.e., running many hyperparameter configurations with a small average budget per trial. Each bracket reduces n by approximately a factor of η until the final bracket, where each trial is given R resources. The total number of resources used in a single execution of Hyperband is $(s_{max} + 1)B$. Thus, for a given budget, Hyperband does approximately $s_{max} + 1$ more work than Successive Halving for a single value of n. The authors recommend repeating the algorithm indefinitely.

Algorithm 4 Hyperband. Adapted from Li et al. (2017)

Require: R, η
$s_{max} \leftarrow \lfloor \log_\eta(R) \rfloor$
$B \leftarrow (s_{max} + 1)R$
for $s = s_{max}, s_{max} - 1, \ldots, 0$ **do**
    $n \leftarrow \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil$
    $r \leftarrow R\eta^{-s}$
    $T \leftarrow$ getHyperparameterConfigurations(n) ▷ Start Successive Halving
    for $i = 0, \ldots, s$ **do**
        $n_i \leftarrow \lfloor n\eta^{-i} \rfloor$
        $r_i \leftarrow r\eta^i$
        $L \leftarrow \text{sample}(t, r_i) \quad \forall t \in T$
        $T \leftarrow \text{topK}(T, L, \lfloor n_i/\eta \rfloor)$
    end for ▷ End Successive Halving
end for
return Configuration with the smallest intermediate loss seen so far.

- getHyperparameterConfigurations(n) - Returns n hyperparameter configurations drawn random uniformly from the hyperparameter configuration space.
- sample(t, r) - Trains the model with hyperparameter configuration t for r epochs and returns the validation loss observed after r epochs.
- topK(T, L, k) - Returns the k top performing configurations from T, based on the corresponding losses L.

3.3.4 BOHB

There is one major disadvantage to both Successive Halving and Hyperband - They draw hyperparameter configurations at random. The BOHB (Falkner et al., 2018a) algorithm proposes a solution to this challenge. BOHB uses Hyperband as its frame, but instead of sampling randomly, it uses an adaptation of the TPE to leverage previous observations. By doing so, the algorithm is able to combine multi-fidelity optimization with SMBO.

The major difference between the KDE BOHB uses and the TPE is that instead of using multiple univariate density estimators, BOHB uses a single multidimensional KDE to 'better handle interaction effects' between hyperparameters. By doing so, BOHB aims to capture dependencies between hyperparameters, which practice has shown exists (Goodfellow et al., 2016; Feurer and Hutter, 2019; Klein and Hutter, 2019). To distinguish between the two KDEs, we will refer to the originally proposed one (Section 3.3.2) as the *univariate TPE* and the one proposed by BOHB as the *multivariate TPE*. For more details on the multivariate TPE, please see Falkner et al. (2018b, Section D).

A challenge with building the surrogate model is that the Hyperband algorithm evaluates the objective at different budgets. At first, many samples evaluated at

small budgets will be available, and as the optimization progresses, more samples evaluated at larger budgets will be observed. BOHB accounts for this by effectively building a model for each intermediate budget r_i used in the Successive Halving runs.

However, the goal of the optimization is to minimize the validation loss with the full budget available. Thus, BOHB only uses the model for the largest budget with at least N_{min} observations (line 4). As the optimization progresses, conclusions are drawn from incrementally higher fidelities and avoid (potentially) misleading conclusions drawn from lower fidelities. BOHB sets N_{min} to equal the number of hyperparameters being optimized plus one.

Line 1 in Algorithm 5 samples uniformly at random a constant fraction ρ of configurations to ensure that BOHB keeps the theoretical guarantees of Hyperband. Line 4 of Algorithm 5 initializes the algorithm with $N_{min} + 2$ random configurations.

Algorithm 5 BOHB sampler. Adapted from Falkner et al. (2018a)

Require: Observations D, fractions of random runs ρ, percentile q, number of configurations N_s, minimum number of observations to build a model N_{min}

1: **if** $rand() < \rho$ **then**
2: **return** random configuration
3: **end if**
4: $b \leftarrow \text{argmax}\{D_b : |D_b| \geq N_{min} + 2\}$
5: **if** $b = \emptyset$ **then**
6: **return** random configuration
7: **end if**
8: Fit multivariate TPE
9: Draw N_s configurations from surrogate model
10: **return** configuration with the highest expected improvement

3.3.5 Exploration vs. Exploitation

An important aspect of any optimization, and particularly SMBO, is the *exploitation-exploration trade-off*; Do we wish to sample hyperparameter configurations that are expected to slightly improve the validation loss (exploitation), or do we wish to sample configurations that might drastically improve the validation loss but also might not improve it at all (exploration)?

With the expected improvement criteria as proposed in Equation 3.14, there is no particular way to control this trade-off. Adding a *trade-off parameter* could be done, but this would only in turn introduce yet another hyperparameter, and is therefore often left out.

The trade-off is however affected by the uncertainty in the surrogate model used, which with both the univariate and multivariate TPE corresponds to the standard deviation, or bandwidth, of the KDEs. With few samples, the distance between neighbours is higher, and thus the bandwidth is higher. This in turn means that at the beginning of the optimization process, hyperparameters are less good/bad in a

larger part of the configuration space, and as the number of samples increases, they become better/worse in a smaller part of the configuration space.

3.3.6 Robustness of hyperparameters

As Definition 3.1 states: Given a dataset $\mathcal{D}$, our goal is to find

$\boldsymbol{\lambda}^* = \text{argmin}_{\boldsymbol{\lambda} \in \Lambda} \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathcal{D}}[V(\mathcal{A}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})]$. However, this $\boldsymbol{\lambda}^*$ might not be the best configuration in the sense that it might not be robust.

First, the dataset $\mathcal{D}$ will change as more data is sampled. Generally, more data should lead to better training loss and generalization. However, the HPO is run with a fixed dataset, and it might just be that $\boldsymbol{\lambda}^*$ performed especially well for exactly this $\mathcal{D}$.

Furthermore, there are some stochastic processes involved in training a neural network. First, the weights are initialized at random. That gives different starting points for the gradient descent algorithm. One configuration might work excellent from one starting point, while it might perform terribly from another point. Secondly, the examples in the mini-batches are sampled at random in each epoch. That also influences how the GD method is able to converge.

When selecting a hyperparameter configuration, we want it to be robust to these changes in data and stochastic processes. For a given hyperparameter configuration $\boldsymbol{\lambda}_1$, with validation loss $f(\boldsymbol{\lambda}_1) = v_1$, we expect reasonably close neighbours of $\boldsymbol{\lambda}_1$ to achieve a similar loss to v_1. However, the dataset used, and the stochastic processes involved might affect this expectation. As mentioned earlier, it might be that $\boldsymbol{\lambda}^*$ was just the result of a lucky combination of hyperparameters and random samples.

One advantage of the model-based algorithms, not mentioned so far, is that they are somewhat able to model the robustness as well. If one trial finds $f(\boldsymbol{\lambda}_1) = v_1$ to perform well, the acquisition function will at one point most likely sample neighbours of $\boldsymbol{\lambda}_1$. If these neighbours perform poorly, either because the hyperparameter configuration is different from $\boldsymbol{\lambda}_1$, or because of the stochastic processes, the surrogate will model this neighbourhood as a bad-performing neighbourhood. Configurations sampled later on will therefore more likely come from outside the neighbourhood of $\boldsymbol{\lambda}_1$, and thus the robustness of the hyperparameter configuration is incorporated in the surrogate model.

4 | Case description

This chapter describes the case and the neural network used in the case. First, a brief introduction of the oil & gas production process is given to provide some context, but the explanation is kept short since understanding the production process is not essential to this work. Then, the dataset used, the structure of the neural network, and particularities of the training and regularization of the network are described.

4.1 Case Context

The residual neural network that is to be optimized is used as a Virtual Flow Meter in oil & gas production. A virtual flow meter is a computational device that based on sensor input estimates the total flow of water, oil, and gas through a production pipe. A simple illustration of the production process is given in Figure 4.1. As illustrated in the figure, all three materials are pumped up from the reservoir, and go through a choke valve that is controlled by an operator, before it flows to a separator that separates the materials from each other.

PWH denotes the pressure at the wellhead, or the upstream pressure. *TWH* denotes the temperature at the wellhead, or the upstream temperature. *CHK* denotes the opening of the choke valve. *PDC* denotes the pressure downstream the choke. *QGL* denotes how much gas lift has been used while producing. *FGAS* denotes the fraction of gas to total flow. *FOIL* denotes the fraction of oil to total flow. *QTOT* denotes the total flow rate.

The VFM is called *NeuralCompass* (NC), and the rest of this chapter describes the foundations of the NC.

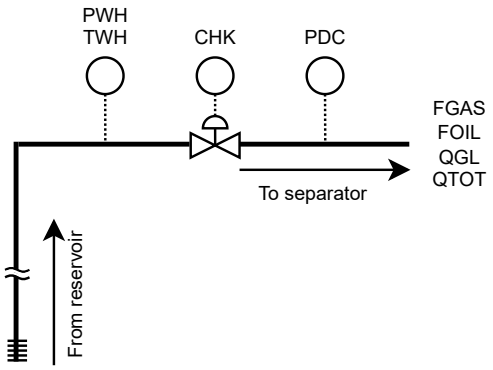


Figure 4.1: Water, oil, and gas is pumped up from the reservoir, goes through an operator-controlled choke valve, and are led to a separator. Adapted from Grimstad (2020).

4.2 Dataset and Preprocessing

Solution Seeker has acquired data from multiple wells, in multiple fields, by multiple instruments. This proposes a challenge, as measurements are different between wells and fields. In addition, since it is assumed that all data comes from the same data generating process, using data from different wells might lead to learning difficulties. The company has developed a learning algorithm and method for preprocessing the data such that a neural network can learn from data across different wells and fields.

Solution Seeker's preprocessing does some automatic cleaning of the dataset and removes some faulty examples. In addition, it performs scaling of the features and targets to appropriate ranges. The features and targets are however not normalized nor standardized. Pre-experimentation with NeuralCompass' standard hyperparameters showed that some further data cleaning was needed. With the original data set provided by Solution Seeker, a typical training with standard hyperparameters would like the blue line in Figure 4.2a with avalanches in the training loss. A corresponding typical validation would look like the blue line in Figure 4.2b, with quite noisy validation loss.

First, it must be stated that by using different hyperparameters, the training and validation loss depicted in Figure 4.2 would look different, and could result in something more similar to the expected exponential decay. However, as the standard hyperparameters have achieved good results before with similar datasets, the NC should not be performing as poorly as it does with this dataset. Figure 4.2 illustrates that the network is not learning anything at all, and the loss comes down to randomness. Correcting that behaviour was the first goal of the data cleaning.

The blue line in Figure 4.2b shows on average some exponential decay, but throughout the training process the noise is loud, and some very large spikes are present. This makes it difficult to use any sort of model-based optimization algorithm, multi-fidelity technique or metaheuristic algorithm from the literature review (Section 2.1). For the model-based, if the surrogate model is fit with a sample where validation loss is at a spike, it would give a false impression of the performance of the hyperparameter configuration under evaluation. For the multi-fidelity techniques, comparing two trials, where at least one validation loss is at a spike, would also lead to an unfair comparison, which might lead to the wrong trial being terminated. For the metaheuristic algorithms, any sort of fitness measured could be misleading as well. In addition, spikes and loud noise in the validation loss might indicate that the network is only memorizing the training set, and not generalizing as is desired. Thus, the second goal of the data cleaning was to reduce the noise and remove the spikes from the validation loss.

While attempting to achieve the two goals as described above, it was also desired to remove as few examples as possible. Therefore, first, all examples with features or targets that indicated that a well was either not producing, or in the process of stopping production, were removed. Outliers for specific wells were then identified by using box plots, as illustrated in Figure 4.3a. Before removing the identified outliers, it was manually investigated if the outlier could be a natural outlier, i.e., the

outlier could be an accurate measurement, by comparing the identified outlier with examples with similar features from the same well. If the outlier could be a natural outlier, the corresponding example was not removed. Removing the outlier shown in Figure 4.3a results in a more informative set of targets, shown in Figure 4.3b. After outliers from specific wells were removed, all wells with less than 25 total examples were removed. This process was repeated by removing the most extreme outliers at first and gradually removing less extreme outliers, to find a suiting trade-off between the smoothness of the loss functions, and the number of examples. After cleaning the original data set, a typical training with standard hyperparameters would like the orange line in Figure 4.2a. A corresponding typical validation would look like the orange line in Figure 4.2b. The number of examples in the original and cleaned dataset, and the number of examples removed, is given in Table 4.1. The table also shows how the dataset is split into a training, validation and test set.

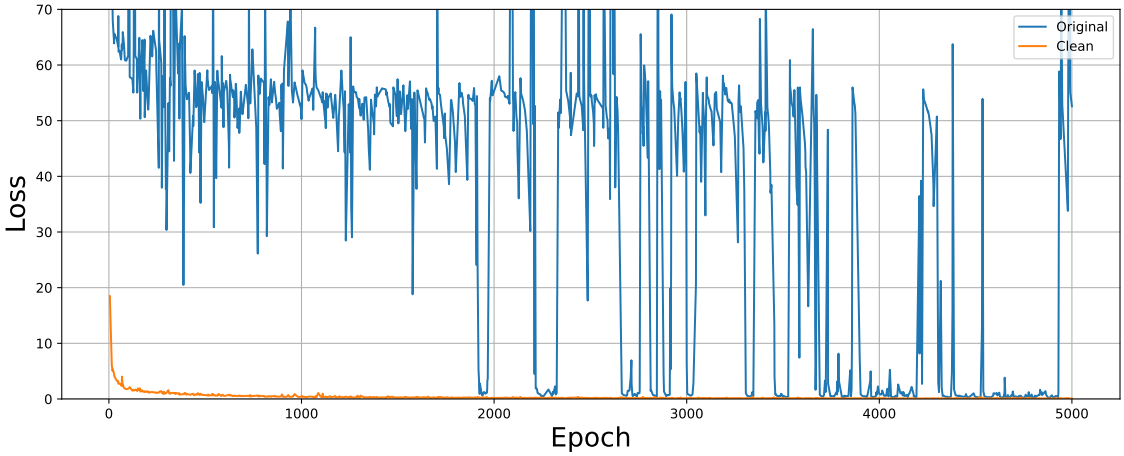
As stated in Section 3.3, the goal of hyperparameter optimization is to extract the optimal out of a selected architecture, where data has been preprocessed and features are already selected. In other words – there are many predefined constraints.

Other aspects of the dataset, such as, e.g., how many wells and fields there are in it, differences between fields, wells within the same field, and wells from different fields, if the data has been anonymized, which instruments that were used for measurement, the ranges for the different features and target, when the different examples were sampled etc are therefore not particularly relevant for the hyperparameter optimization in itself, because they fall within those predefined constraints.

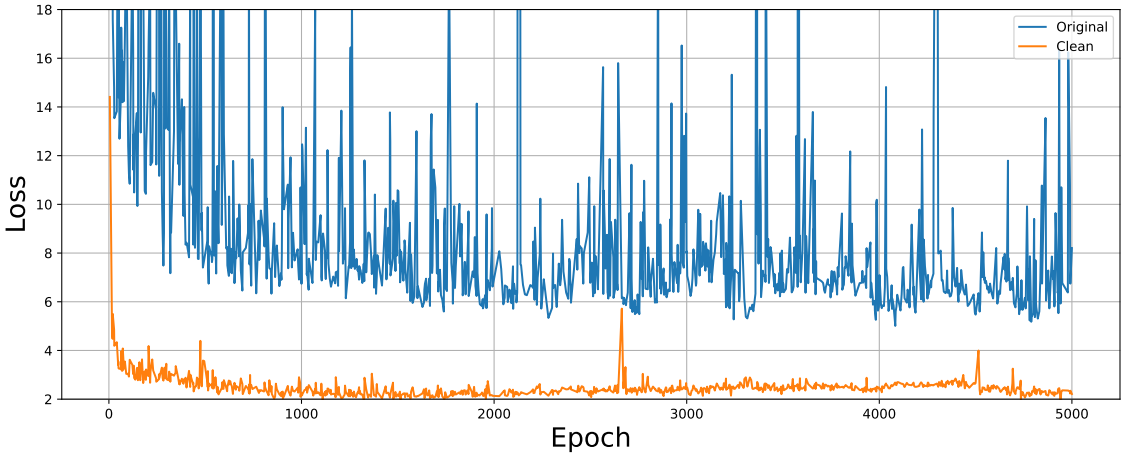
Nevertheless, it should be stated that changing the preprocessing and feature selection might also improve the performance of the neural network, as well as the performance of the HPO-algorithms. However, it falls outside the scope of this work.

Table 4.1: Overview of the dataset used in the case, and how it is split into a training, validation, and test set. The numbers correspond to the number of examples, except the bottom row, which is given in per cent.

	Total	Training	Validation	Test
Original	80138	62252	13761	4125
Clean	79356	61690	13574	4092
Removed	782	562	187	33
Removed[%]	0.976	0.903	1.359	0.800

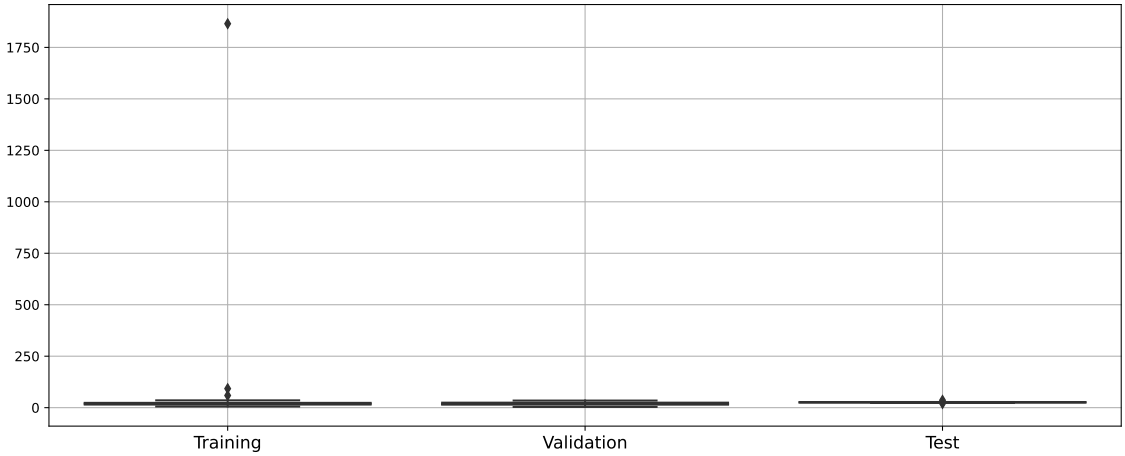


(a) Training loss.

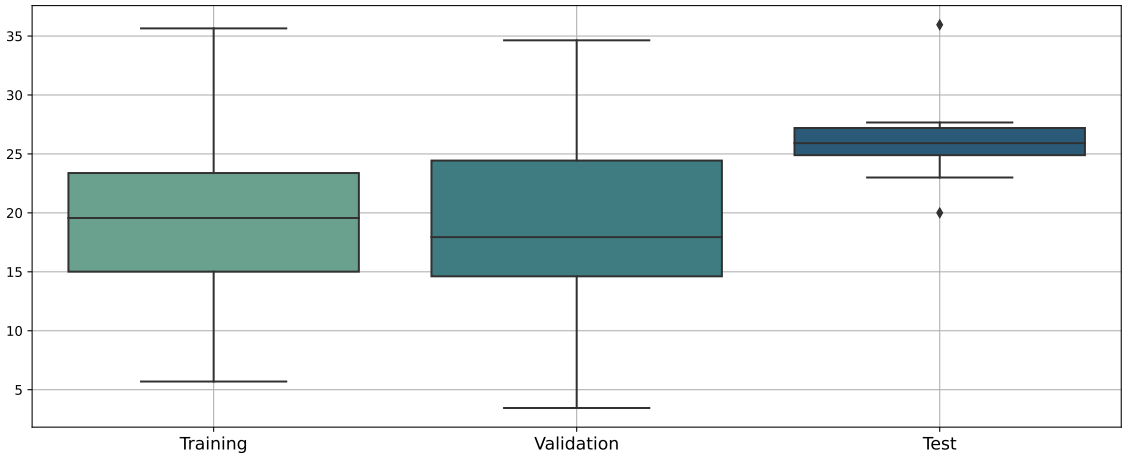


(b) Validation loss.

Figure 4.2: Training and validation loss with original and cleaned dataset. The blue line illustrates the typical development of loss with the standard hyperparameters on the original dataset, and the orange line illustrates the typical development of loss with the standard hyperparameters on the cleaned dataset. Cleaning the data resulted in a significantly better and smoother training and validation loss.



(a) Original data.



(b) Cleaned data.

Figure 4.3: Box plots of the target in the training, validation, and test set. Whiskers extend to $1.5 \times IQR$, and outliers are drawn as diamonds. The grey solid horizontal line marks the median. Notice the different y-scale between **(a)** and **(b).** Removing the extreme outlier in the training set in **(a)** results in the new distribution in **(b)**.

4.3 Structure

As mentioned in Section 4.2, the data set comes from multiple wells, in multiple fields, and is measured by multiple instruments. The production process of each well is unique, and thus the data obtained for each well is also unique. However, the overall process (Figure 4.1) is still the same, and it is therefore assumed that the data from each well comes from somewhat similar distributions. The NC utilizes Multitask learning (MTL) (Ruder, 2017) in a comparable – but not identical – way of Sandnes et al. (2021) to improve '*generalization by leveraging the domain-specific information in the training signals of related tasks*' (Caruana, 1997). In this context, task refers to wells, where each well is a unique task.

The NC uses 7 features from the data set: PWH, TWH, CHK, PDC, FGAS, FOIL and QGL as described in Section 4.1. To accommodate for MTL, task-specific learning parameters are used alongside the features. Consider, e.g., when a single example is forward propagated through the network. This single example, obviously, only comes from a single well. At the input layer, a set of task-specific learning parameters are added in parallel with the features. These parameters are deactivated if the example does not come from the corresponding task, and are otherwise equal to a normal learning parameter. Thus, when the cost is backwards propagated, the task-specific learning parameters are updated.

By adding the task-specific learning parameters, the NC can learn a generalized production process by tweaking the general learning parameters, and take task-specific information into account by tweaking the task-specific parameters. In other words, the task-specific learning parameters provide some context to the training and are thus referred to as context parameters. Exactly how these context parameters are added is beyond the scope of this work. The input layer of the NC has a width equal to the number of features, plus the number of context parameters.

The hidden layers of the network are residual blocks as described in Section 3.2.5. Note that the residual blocks are specially designed, and do not necessarily look like the one in Figure 3.10. Each residual block in the NC consists of multiple fully connected layers with equal widths. The width of the network thus refers to the width of these fully connected layers. The depth of the network is determined by the number of residual blocks used.

The NC further uses QTOT as described in Section 4.1 as the target. The output layer thus maps the activation from the last residual block to a single flow rate estimation $\hat{y}$. A simple illustration of the NC with three residual blocks is given in Figure 4.4.

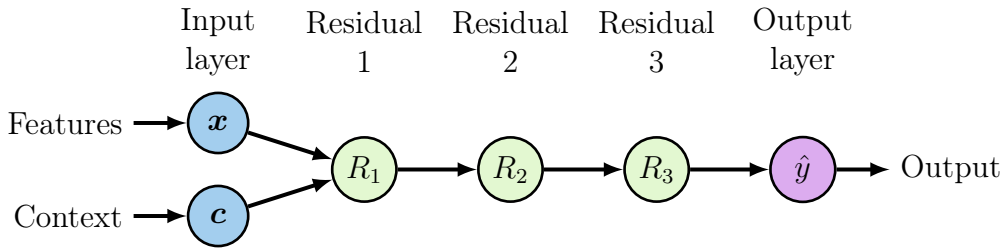


Figure 4.4: Illustrative structure of the NeuralCompass, showing how the context-specific learning parameters and residual blocks are used.

4.4 Training and Regularization

Activation function

The fully connected layers in the residual blocks use the *Rectified Linear Unit* (ReLU) activation function (Nair and G. E. Hinton, 2010): $a(x) = \max\{0, x\}$. There exists many other activation functions (Nwankpa et al., 2018), but the default recommendation is to use the rectified linear unit function: Goodfellow et al. (2016) describes why: '*These units are easy to optimize because they are so similar to linear units. The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active. The gradients are not only large, but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.*' ReLU is illustrated in Figure 4.5.

The output layer uses a linear activation function, which simply applies a linear transformation to the incoming data: $\hat{y} = A\boldsymbol{z} + \boldsymbol{b}$. Thus, a single estimation of the total flow rate is produced.

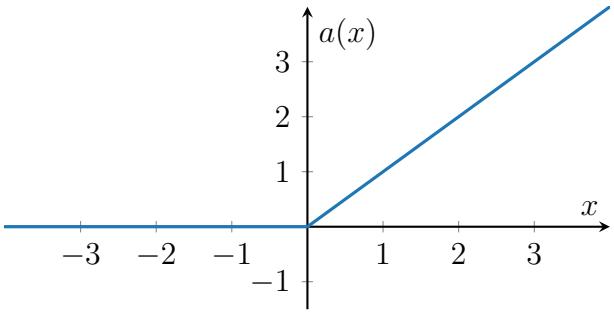


Figure 4.5: Rectified Linear Unit activation function.

Weight initialization

The NC uses an adaptation of *He-initialization* (He et al., 2015) since the neurones in the hidden layers of the network use the ReLU activation function. He-initialization was specially designed for the ReLU activation function and its derivate properties. For a given layer, it draws weights from a Gaussian distribution:

$$\sim \mathcal{N}(0, \frac{gain^2}{N}),$$

where N is the number of inputs to the layer. The gain is set to $\sqrt{2}$ for the nodes in the layers that use the ReLU activation function, and 1 for the nodes in the output layer since the activation function is a linear transformation. These values are standard values. The weights are furthermore scaled to maintain unit variance throughout the network, i.e., if all features are standard normal distributed, then the output of the initialized model will also be standard normal distributed.

Optimizer

Section 3.2.2 introduced the standard gradient descent optimization method, along with the stochastic and mini-batch versions of it. There exist different versions of the gradient descent optimizer (Ruder, 2016), and the NC uses *Adam* (Kingma and Ba, 2014) as its optimizer. The name Adam comes from *adaptive moment estimation*, and combines the use of *momentum* and *adaptive learning rates*.

Momentum simply introduces a short term memory of previous steps, which allows the algorithm to build inertia in some direction in the search space, in order to overcome noisy gradients and avoid local minima:

$$\begin{aligned} \boldsymbol{m}_{k+1} &= \beta \boldsymbol{m}_k + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k; \boldsymbol{x}, \boldsymbol{y}) \\ \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \boldsymbol{\alpha}_k \boldsymbol{m}_{k+1}, \end{aligned} \tag{4.1}$$

where β is the momentum parameter. Setting $\beta = 0$ results in standard gradient descent. See Goh (2017) for a beautiful interactive explanation of why and how momentum works.

Adaptive learning rates is a technique that adapts the learning rate individually for each learning parameter as the training goes on. Adam finds its adaptive learning rate inspiration from *RMSProp* (G. Hinton et al., 2012), which individually adapts the learning rates of all model learning parameters by scaling them inversely to an exponentially weighted moving average of the square root of the sum of their historical squared values within a defined window. With Adam, you then have a defined maximum learning rate, α_{max}, for all learning parameters, and individually adapted learning rates in $\boldsymbol{\alpha}$, such that the learning rate at step k, $\boldsymbol{\alpha}_k$, is maximum α_{max}.

By combining an adaptation of RMSProp with momentum, Adam is regarded as a reasonably good go-to optimizer, recommended by e.g., Ruder (2016). Practice has

also shown that Adam is 'fairly robust to the choice of hyperparameters' (Goodfellow et al., 2016), but there is no consensus on which optimizer is the best.

Learning rate scheduler

The NC also uses a *learning rate scheduler*. Learning rate scheduling is somewhat similar to an adaptive learning rate, only that it affects all parameters similarly instead of individually. Combining Adam with a learning rate scheduler is the same as saying that the maximum learning rate for all parameters, α_{max}, is adjusted by the scheduler throughout the training. The individual learning rates at step k, $\boldsymbol{\alpha}_k$, can therefore be lower if the optimizer decides so.

The NC uses a so-called stepping learning rate scheduler, which reduces the learning rate by a factor every time the training reaches a predefined milestone. The milestones were defined as every 100th epoch starting from and including the 4000th epoch, and the factor was set to 0.6. The reason behind using the learning rate scheduler in the late stage of the training is to stabilize the training and validation loss, and to hopefully allow the optimizer to close in on a good local minimum without limiting the training too much.

Cost function

The NC derives its cost function from the theory of *maximum likelihood estimation* (MLE), which is a way of estimating the parameters of a probability distribution given some observed data. In other words, MLE wishes to maximize the probability of the learning parameters θ describing the true data generating distribution, given the examples observed in the data set:

$$\boldsymbol{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}}\, p(\boldsymbol{x}; \boldsymbol{\theta})$$

For a linear regression problem, such as predicting the flow rate from a set of features, it can be shown that (Goodfellow et al., 2016) maximizing the likelihood yields the same estimate of the optimal parameters θ as minimizing the mean squared error (MSE):

$$MSE = \frac{1}{n} \sum_{i=1}^{n} ||\hat{y}_i - y_i||^2$$

By adding a L^2 parameter norm penalty, or *weight decay*, $\kappa\Omega(\boldsymbol{\theta}) = \kappa||\boldsymbol{w}||_2^2$ to the MSE function we arrive at the cost function

$$J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1}^{n} ||\hat{y}_i - y_i||^2 + \kappa||\boldsymbol{w}||_2^2, \tag{4.2}$$

where n is the number of examples.

It can also be shown that this is equal to performing a *maximum a posteriori* (MAP) point estimate

$$\boldsymbol{\theta}_{MAP} = \underset{\theta}{\operatorname{argmax}}\, p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \underset{\theta}{\operatorname{argmax}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}),$$

with a zero-mean Gaussian prior on the weights (Goodfellow et al., 2016).

Since the NC performs multitask learning as explained in Section 4.3, the cost function in Equation 4.2 has to be updated slightly:

$$J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1}^{k} ||\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i||^2 + \kappa_{shared} ||\boldsymbol{w}_{shared}||_2^2 + \kappa_{specific} ||\boldsymbol{w}^i_{specific}||_2^2, \qquad (4.3)$$

where n are the total number of examples, k are the different tasks, $\boldsymbol{w}_{shared}$ denotes the shared weights, $\boldsymbol{w}^i_{specific}$ denotes the task-specific weights for task i, and κ_{shared} and $\kappa_{specific}$ donates the norm penalty of the shared and task-specific weights respectively.

5 | Method

This chapter describes the method used to perform the case study, and indirectly forms a guide to HPO for Solution Seeker. First interesting hyperparameters of the NC are identified before the selection of HPO-algorithms is discussed. Then the experiment design is explained, before the implementation of the experiment is described.

5.1 Identifying Interesting Hyperparameters

The NC has many hyperparameters, and limiting the configuration space is as described in Section 3.3.1 paramount for a satisfying result. Consider, e.g., which optimizer to use as a hyperparameter. The popular machine learning framework PyTorch (Paszke et al., 2019) offers 13 different optimizers, and just Adam has its own 6 hyperparameters. Limiting the configuration space is as described earlier key, and this section describes why the hyperparameters chosen to be optimized in this work were chosen. A summary is given in Table 5.1.

The learning rate

The learning rate is widely recognized as one of the most – if not the most – important hyperparameter (Goodfellow et al., 2016; Klein and Hutter, 2019). Goodfellow et al. (2016) even writes; '*If you have time to tune only one hyperparameter, tune the learning rate*'. Hence, the learning rate was chosen as the first interesting hyperparameter. Adam uses a standard learning rate of 1×10^{-3}, and the NC has previously been trained with a learning rate of 1×10^{-4}. It was chosen to set the lower limit of the learning rate at 1×10^{-6}, and the upper at 1×10^{-2}, each corresponding to respectively decreasing and increasing the learning rate by a factor of 100 from NC's standard.

The learning rate belongs to $\mathbb{R}^+$, but sampling it uniformly at random within the defined limits is not effective. That is because the learning rate has a multiplicative effect on the gradient descent (Equation 3.6). Changing the learning rate by a delta of 1×10^{-3} of a model originally trained with $\alpha = 1 \times 10^{-1}$ would result in a very small difference, but it for a model originally trained with $\alpha = 1 \times 10^{-4}$ would have a great effect. It was therefore decided to sample the learning rate from a logarithmic uniform distribution (log uniform), to take the multiplicative factor into account.

Capacity and regularization

Secondly, the connection between capacity, regularization, and overfitting & underfitting was identified as an interesting aspect to optimize. This was motived by the following statement from Goodfellow et al. (2016) '*Controlling the complexity* [capacity] *of the model is not a simple matter of finding the model of the right size, with the right number of parameters. Instead, we might find – and indeed in practical deep learning scenarios, we almost always do find – that the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately.*'

There were several ways that these relationships could be altered; Changing the depth, the width, the weight penalization parameters, adding different norm penalty terms, adapting other regularization strategies, changing the structure of the residual blocks, etc. To again limit the configuration space, it was decided to use the depth as a variation of capacity, and the weight penalty parameters in Equation 4.3 as ways of optimizing the regularization.

The NC used a depth of five residual blocks, and the upper limit was set to five, and the lower limit was set to two, yielding a difference of $\pm$ three blocks. The depth was chosen uniformly at random from within this range.

The weight penalization parameters belong to $\mathbb{R}^+$ and have a multiplicative effect just like the learning rate. Thus, it was decided to also sample these from a log uniform distribution. Both parameters were originally set to 1×10^{-9}. The lower limit was decided to be 1×10^{-10}, to have some decrease available. It was not set any lower, as any change below 1×10^{-10} would be insignificant. The upper limit was set to 1×10^{-3} to allow for strong regularization. The limits were set equal for both parameters.

Multitask specific

Since the neural network performs multitask learning, it was finally decided to experiment with the number of context parameters to see how they affect the training. The NC originally used 20 context parameters per task, and it was decided to draw parameters uniformly at random from the following set: $\{1, 2, 3, 4, 6, 8, 11, 14, 17, 20, 25, 30\}$. The set was designed to simulate a semi-logarithmic scale while skipping some steps to further limit the configuration space. In addition, tuning on weight penalty parameter for the context-specific parameters, $\kappa_{specific}$, is also an interesting multitask specific aspect, further justifying that choice.

A summary of all selected hyperparameters and their range and distribution is given in Table 5.1.

Table 5.1: Identified hyperparameters and their defined range, distribution and standard value.

Hyperparameter	Range	Distribution	Standard value
Learning rate	$[10^{-6}, 10^{-2}]$	Log uniform	10^{-4}
Depth	$\{2, 3, 4, 5, 6, 7, 8\}$	Uniform	5
κ_{shared}	$[10^{-10}, 10^{-2}]$	Log uniform	10^{-9}
$\kappa_{specific}$	$[10^{-10}, 10^{-2}]$	Log uniform	10^{-9}
Context parameters	$\{1, 2, 3, 4, 6, 8, 11, 14, 17, 20, 25, 30\}$	Uniform	20

5.2 Selecting HPO Algorithm

It was desired to try at least three different HPO algorithms to compare their performance, one baseline model and at least two challengers. This section explains how and why the selected algorithms were chosen.

Before discussing how the selected HPO-algorithms were chosen, it is important to mention the available hardware. Solution Seeker provided access to a machine setup with 4 CPUs with a total of 15 GB memory, and one NVIDIA Tesla T4 GPU, through the Google Cloud Platform. See Appendix A for more details.

Observations

As mentioned in Section 2.1, to achieve a good result, it is cardinal to choose an appropriate optimization algorithm for your case. Through pre-trials and experimentation, the following was observed regarding the NC:

i. Training for the standard 5000 epochs took around 30 minutes on the provided hardware with standard hyperparameters.

ii. Although it is difficult to say what a big configuration space is, the chosen configuration space is at least not small.

iii. There are no real parallel resources available, as only one GPU is available.

iv. The validation loss is still a bit noisy, even after cleaning the data. The learning rate scheduler reduces the noise over the last 4000 epochs.

v. The configuration space contains both continuous and integer categorical hyperparameter domains.

Item i. underlines that validating a trial is expensive and that validating many trials would take many hours. This speaks in favour of a model-based algorithm, which, if successful, should be able to make smart acquisitions and steer away from the poor

performing configurations. It also speaks in favour of using multi-fidelity techniques, as they are designed exactly for this purpose.

Item ii. simply does not disfavour or favour any algorithm.

Item iii. disfavours the bandit-based and the metaheuristic algorithms, as their capability is greatly reduced when trials are forced to run sequentially. E.g., a standard run of Hyperband would yield 2187 trials running in parallel for 2 epochs. Starting and pausing all of these trials on the same GPU would yield a massive overhead.

Item iv. disfavours the metaheuristic and model-based algorithms as well as the multi-fidelity techniques, because these algorithms make decisions based on the validation loss, as explained in Section 4.2.

Item v. disfavours the use of a Gaussian process-based surrogate model, as these do not natively support integer hyperparameters.

Choosing algorithms

First, the use of metaheuristic algorithms was quickly ruled out as no parallel recourses were available, and because they provide no guarantees for any strong final performance. Nevertheless, a model-free method was desired as a baseline algorithm. It was therefore decided to use the vanilla random search as the baseline algorithm, because of its non-assuming simple nature.

For the first challenger, random search combined with the median stopping rule was chosen to investigate the effect of using multi-fidelity techniques. This was desired because of Item i., and the MSR was intended to speed up the termination of bad performing trials to be able to sample more configurations. Although the bandit-based algorithms Successive Halving and Hyperband have shown better anytime and final performance than random search, they were ruled out due to the lack of parallel resources.

Items i. and iii. all speak in favour of a model-based optimization algorithm, i.e., Bayesian optimization with either Gaussian processes, random forests or tree-structured Parzen estimator, or BOHB. BO-GP was then ruled out because of Item v.. Although BOHB is regarded as a general, well-suited HPO algorithm, it still relies on the Hyperband algorithm. Again, due to the lack of parallel resources, BOHB was therefore deemed less suited for this case. This leaves SMAC and BO-TPE as the most likely well-suited HPO algorithms.

For further challengers, it was as explained above therefore desired to use either BO-TPE or SMAC. In addition to investigating the performance of a model-free method with a model-based method, several other aspects of HPO could be explored:

- SMAC vs BO-TPE.
- The effect of multi-fidelity techniques.

- Comparing different multi-fidelity techniques for one SMBO algorithm.
- Using the univariate TPE or the multidimensional TPE.

BO-TPE obtained a stronger final performance than SMAC on datasets that are more similar to our application in Klein and Hutter (2019), and having to start somewhere, it was therefore chosen to not compare SMAC and BO-TPE. In other words, BO-TPE was chosen as the preferred algorithm. Investigating the performance of different multi-fidelity techniques was furthermore not desired due to the lack of parallel resources.

This left the choice between either investigating the effect of multi-fidelity optimization by running one optimization process with vanilla BO-TPE and another with some early-stopping technique, or running one process with the univariate TPE and one with the multivariate TPE.

Looking at the latter option first. Intuitively, it makes sense that there are dependencies between hyperparameters. Literature also shows this (Goodfellow et al., 2016), as e.g., deeper models often require stronger regularization than shallower models. Unfortunately, there does not, to the best of our knowledge, exist much literature comparing the univariate TPE to the multivariate one. Preferred Networks, the company that makes the HPO-framework Optuna (Akiba et al., 2019) compared in Hiroyuki (2020) the performance between the two methods on the benchmark in Klein and Hutter (2019) when they updated their library to also include the multivariate estimator. Hiroyuki (2020) reported that the multivariate TPE outperformed the univariate one. Klein and Hutter (2019) furthermore reported on the importance of hyperparameter pairs in their findings and showed that some pairs were far more important than others. Furthermore, our configuration space is fairly similar to the one BOHB (Falkner et al., 2018a) used, where the multivariate estimator outperformed the univariate one. It was therefore believed that the multivariate estimator would suit this case better.

Looking at the first option, combining BO-TPE (or any other SMBO method) with early stopping, we can compare both vanilla random search with vanilla BO-TPE, and random search with early stopping and BO-TPE with early stopping. In addition, combining BO-TPE with early stopping results in an interesting question; Since some trials are early stopped, the surrogate model will be partly built by intermediate results. How will this affect the acquisition and convergence of the optimization algorithm? This is furthermore interesting in the sense of performing hyperparameter optimization within a restricted time budget, as the goal of early stopping is to speed up the convergence of the optimization. This is in accordance with the primary object of this work.

Based on these two considerations, it was decided to use the multivariate TPE instead of the univariate, and investigate the effect of combining the BO algorithm with early stopping. Again, the MSR was chosen as the multi-fidelity technique.

5.3 Experiment

The following HPO algorithms were as described in Section 5.2 chosen:

Table 5.2: Selected HPO-algorithms

Abbreviation	Algorithm
RS	Vanilla random search.
RS-MSR	Random search with median stopping rule.
BO	Vanilla multivariate BO-TPE.
BO-MSR	Multivariate BO-TPE with median stopping rule.

The network was allowed to train for at most 5000 epochs, with a mini-batch size of 2048 examples.

All algorithms were given the standard hyperparameters as the first trial to run. Both BO and BO-MS3 were initialized with an additional 9 random configurations before the surrogate model was used to suggest configurations. This was done to initialize the surrogate model with varying samples throughout the configuration space.

The MSR was implemented with a grace period of 2500 epochs, such that trials could at the earliest be stopped after 2500 epochs. A minimum number of 50 trials were required before the first trial could be stopped. The same MSR was used in RS-MSR and BO-MS3.

The validation loss was measured with the average mean square error between the prediction and target for all examples in the validation set of the 10 latest epochs:

$$v = \frac{1}{10} \sum_{e=E-9}^{E} \left(\frac{1}{N} \sum_{n=1}^{N} (y_n^e - \hat{y}_n^e)^2 \right), \tag{5.1}$$

where E is the last epoch, N is the number of examples in the validation set, y_n^e is the target for example n in epoch e, and $\hat{y}_n^e$ is the prediction for example n in epoch e.

This was done to smoothen the validation loss a bit further, and implies that both the surrogate model and early stopping, were based on this metric. This was done such that the early stopping and surrogate models partly constructed with early-stopped samples would not be influenced as much by the noise in the loss. Furthermore, it was expected that the difference between the MSE at trial E and the loss of Equation 5.1 would be quite small in the ultimate stages of training because of the learning rate scheduler. Note that this in turn also means that the running average used to get the mean in the MSR is also based on the validation loss of Equation 5.1. For the first nine epochs, the average was calculated up until the number of epochs passed.

Computational budget

Part of the primary objective of this work is to perform hyperparameter optimization within a restricted budget, both time and money-wise.

Originally, it was desired to run each method for at least 200 trials, to be able to cover the configuration space enough. One trial used around 30 minutes, which with 200 trials corresponds to approximately 4 days and 4 hours. To account for other aspects, such as writing experiment data to logs, interruptions that might occur, and other unknowns, another 5 hours were added to the experiment. Thus, each method was limited to run for 4 days and 9 hours.

To fit within Solution Seeker's expense budget, each experiment was method to run 3 times, and one computer was made available for each method. Thus, the total run time would be approximately 13 days. Running each method 3 times also gives some indication of performance variance for the different algorithms. This fits with the objective of this work, which was not to find the best configuration, but rather to acquire knowledge and insight into the suitability of the different methods. It was considered running the experiment more times, but the costs of obtaining a more statistically robust result were deemed too high.

5.3.1 Hypothesis

Based on the literature, theory and selected methods, the following hypothesis was formed for the experiment:

H1: All methods should find better-performing hyperparameters than the standard hyperparameters.

H2: Adding early stopping to a method should yield better anytime and final performance than the same method without early stopping.

H3: Vanilla multivariate BO-TPE should have better final performance, and similar or better anytime performance, than vanilla random search. Similarly, Multivariate BO-TPE with MSR should have better final performance, and similar or better anytime performance, than random search with MSR.

5.4 Implementation

The neural network itself was implemented in PyTorch (Paszke et al., 2019) and was provided by Solution Seeker. One major modification was made to speed up the training process: The network was adapted to use PyTorch's *Automatic Mixed Precision*, which leverages Tensor Cores to significantly speed up training. It does so by using mixed precision for different operations, and by so accelerating throughput while maintaining accuracy. The width of the fully connected layers in the residual

blocks was altered to be a multiple of 8 to accommodate for the use of automatic mixed precision.

The hyperparameter optimization was realized through Ray Tune (Liaw et al., 2018). It was desired to use Ray Tune as the author was confident with the library, and because a variety of HPO algorithms are available if new methods are to be explored in further works. RS and RS-MSR were performed using Tune's *BasicVariantGenerator*. BO and BO-MSR were realized through Tune's *OptunaSearch* and Optuna's *TPESampler* (Akiba et al., 2019). The standard values of the *TPESampler* were used. The MSR was implemented using Tune's *MedianStoppingRule*. A quick overview of the Python implementation of the selected algorithms (Table 5.2) is given in Appendix B. The specific python implementation is straightforward, and we refer the reader to Liaw et al. (2018), Akiba et al. (2019) and Paszke et al. (2019) for introduction to the aforementioned libraries.

The experiment was ultimately deployed with Docker (Merkel, 2014), to make it easy to develop locally and run remotely. It also allowed for an easy way of letting processes run on the remote machine without requiring SSH-connection. The experiment was as mentioned run on the Google Cloud Platform with the hardware configuration given in Appendix A.

6 | Results

This chapter presents the results of the experiment.

The loss refers to the loss used in the optimization process, Equation 5.1, which is the average validation loss over the final 10 epochs. This was done to smoothen the validation loss a bit, as described in Section 5.3.

The average loss of the standard hyperparameters was obtained from the first trial of each run of each method, as these trials as mentioned all used the standard hyperparameters. The average was calculated to be 2.46, and is referred to as $\bar{v}$. This number will be used as a benchmark. Δ signifies the improvement, i.e., the decrease in loss, from $\bar{v}$.

6.1 Final solutions

The best loss found from each run of each method, and the corresponding improvement from $\bar{v}$, Δ is given in Table 6.1. The table also show at which trial the loss was found at, along with what wall clock time since the run was started it was found at. The average loss and Δ of each method are also given in Table 6.1. The best run is highlighted in bold text, and the best average is highlighted in italics.

As can be seen from Table 6.1, BO was able to find both the best individual score and also obtained the best average. The best score was found after 193 trials, or after 3 days, 17 hours and 50 minutes, corresponding to about 86% of the allowed time. The best loss obtained was 1.44, corresponding to a Δ of 1.03 or around 42% lower than $\bar{v}$. The best average was 1.53, corresponding to a Δ of 0.94 or around 38% lower than $\bar{v}$. The worst average belongs to RS, with an average loss of 1.64, corresponding to a Δ of 0.82 or around 33%. Every run of every method was able to improve from $\bar{v}$.

Table 6.1: Results of experiment. The table shows the best loss found at each run for every method and the corresponding average. Δ is the improvement from the average of the 12 initial runs with standard hyperparameters (2.46). Δ[%] shows the same improvement in per cent. Finally, the trial and wall clock time that the best configuration was found at is shown. The best configuration is highlighted in bold text, and the best average is highlighted in italics.

Method	Run	Loss	Δ	Δ[%]	At trial	After wall clock time
RS	1	1.52	0.95	38.45	110	2 days, 8 hours, 48 minutes
	2	1.63	0.84	34.07	73	1 day, 13 hours, 23 minutes
	3	1.79	0.68	27.60	43	20 hours, 12 minutes
	AVG	1.64	0.82	33.37		
RS-MSR	1	1.53	0.94	38.15	12	5 hours, 40 minutes
	2	1.59	0.88	35.62	190	3 days, 16 hours, 9 minutes
	3	1.69	0.78	31.50	182	3 days, 7 hours, 52 minutes
	AVG	1.60	0.87	35.09		
BO	**1**	**1.44**	**1.03**	**41.55**	**193**	**3 days, 17 hours, 50 minutes**
	2	1.53	0.94	38.01	238	4 days, 7 hours, 53 minutes
	3	1.62	0.85	34.51	93	1 day, 21 hours, 37 minutes
	AVG	*1.53*	*0.94*	*38.02*		
BO-MSR	1	1.57	0.90	36.48	181	4 days, 4 hours, 10 minutes
	2	1.55	0.92	37.40	113	2 days, 9 hours, 52 minutes
	3	1.62	0.85	34.26	107	2 days, 3 hours, 11 minutes
	AVG	1.58	0.89	36.04		

Figure 6.1 shows which methods found the configurations in the top 1, 5 and 15 percentile out of the total, 2588 trials. BO was able to find the most configurations in all percentiles, thus finding most of the best and most of the good configurations. RS, RS-MSR and BO-MSR performed fairly similar.

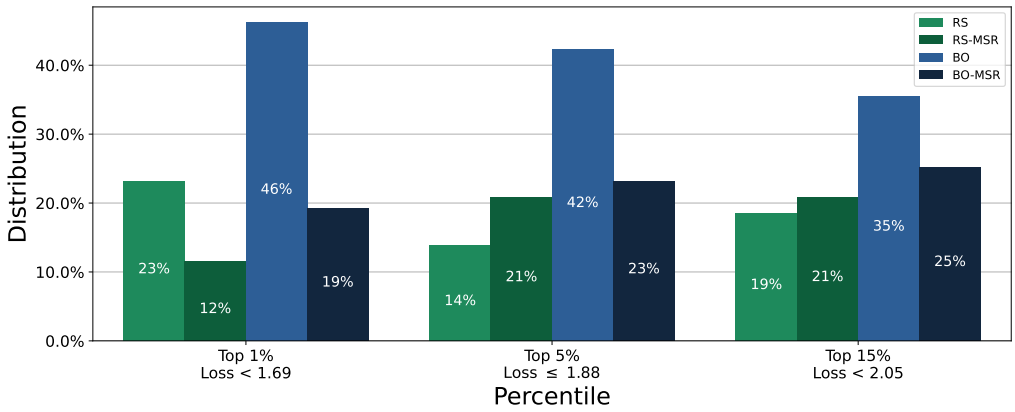


Figure 6.1: The plot shows which methods found the configurations in the top 1, 5 and 15 percentile out of the total, 2588 trials. Especially BO was able to find many good configurations.

6.2 Runtime performance

Performance bottleneck

As you can see from Figures 6.3 and 6.4, RS-MSR was only able to run a few more trials than RS within the defined time window, even though as Table 6.2 shows, at least 80 trials were early stopped in each run. Furthermore, BO-MSR ran fewer trials than BO, as can be seen from Figures 6.5 and 6.6, even though at least 20 trials were early stopped in each run as shown in Table 6.3.

Based on this, the specific implementation of the MSR used in this experiment was deemed a performance bottleneck. As of 16th June 2022 it is not exactly clear why this bottleneck occurred. It might be because of Tune's implementation of the MSR and scheduling in general, how pausing and possibly continuing each epoch affects the efficiency of the GPU, or other unknown variables.

Nevertheless, it is still possible to get some insight into the effect of early stopping.

Runtime performance

The average cumulative minimum of each method is given in Figure 6.2, and the cumulative minimum of each run for each method, along with the 90% confidence interval around the mean, is given in Figures 6.3 to 6.6. As the figures show, all methods were on average able to quickly improve from $\bar{v}$ as can be seen in Figure 6.2. Each run was also able to improve quickly from $\bar{v}$ as can be seen in Figures 6.3 to 6.6.

Figure 6.2 shows that both the any-time and final performance for the average of each method is fairly similar, with a slightly better final performance of BO. Figures 6.3 to 6.6 shows that the confidence intervals (CI) of BO and BO-MSR are much smaller than the CI of RS and RS-MSR for the majority of the time.

Figures 6.3 and 6.4 shows that both RS and RS-MSR made large steps at the beginning of the run, but no significant improvements were observed in the later stages. Figures 6.5 and 6.6 shows that both BO and BO-MSR also made large steps in the beginning, but their development is a bit more incremental than the avalanches observed for RS and RS-MSR.

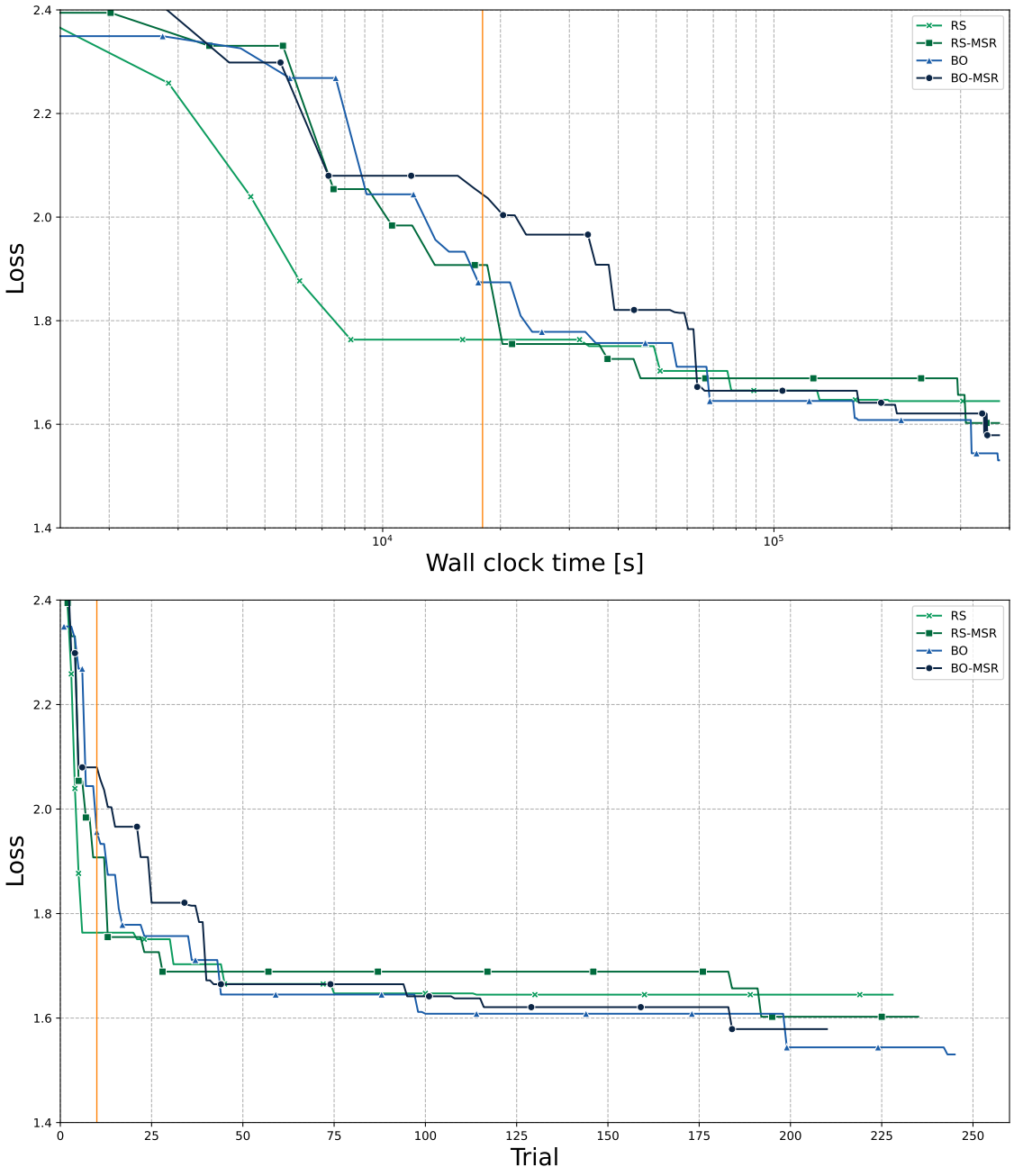


Figure 6.2: Average cumulative minimum for all methods. The orange vertical line marks where BO and BO-MSR stopped sampling randomly. There is no significant difference between any of the methods when it comes to both anytime and final performance.

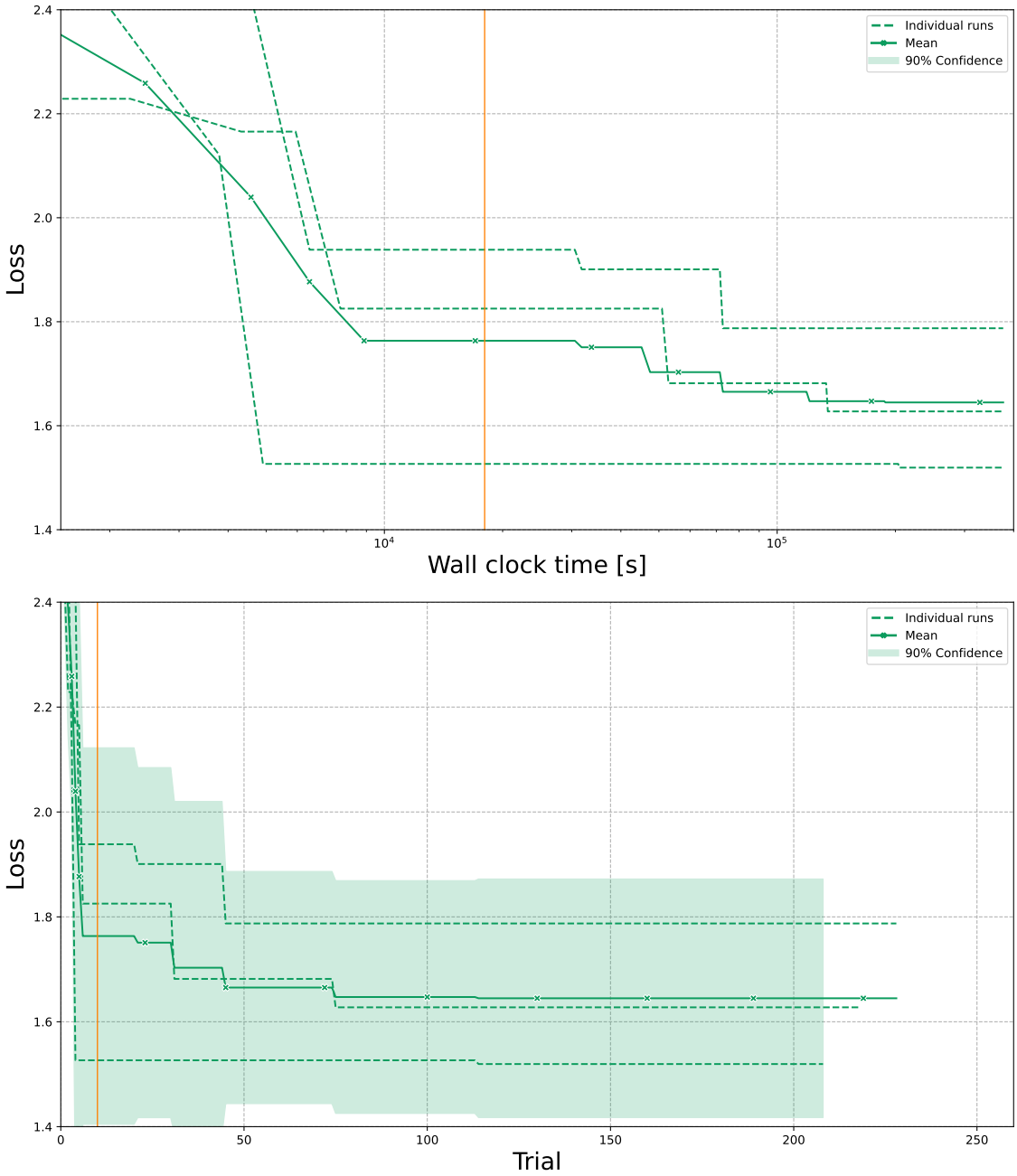


Figure 6.3: Cumulative minimum for RS. The orange vertical line marks where BO and BO-MSR stopped sampling randomly. The confidence interval stretches out from the mean and is only shown where all three runs were still sampling. One run was able to find a relatively good configuration quite fast, resulting in an initially large confidence interval, and remains quite large throughout. No significant improvements were found after around 8×10^4 seconds, or 50 trials.

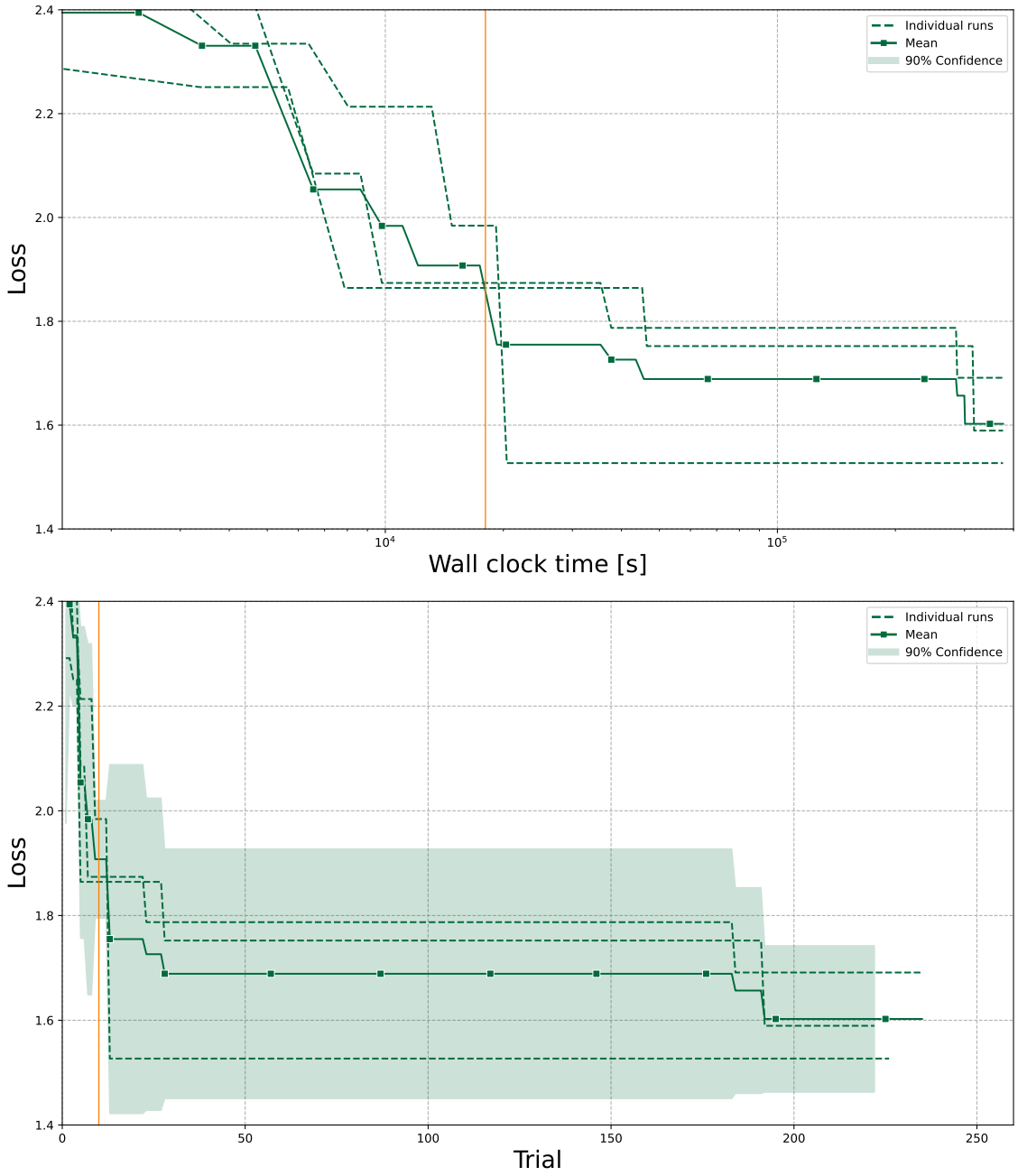


Figure 6.4: Cumulative minimum for RS-MSR. The orange vertical line marks where BO and BO-MSR stopped sampling randomly. The confidence interval stretches out from the mean and is only shown where all three runs were still sampling. All three runs evolve quite similar up until around 2×10^4 seconds, or 10 trials, where one run improved significantly. That results in a large CI, which reduces a bit towards the end.

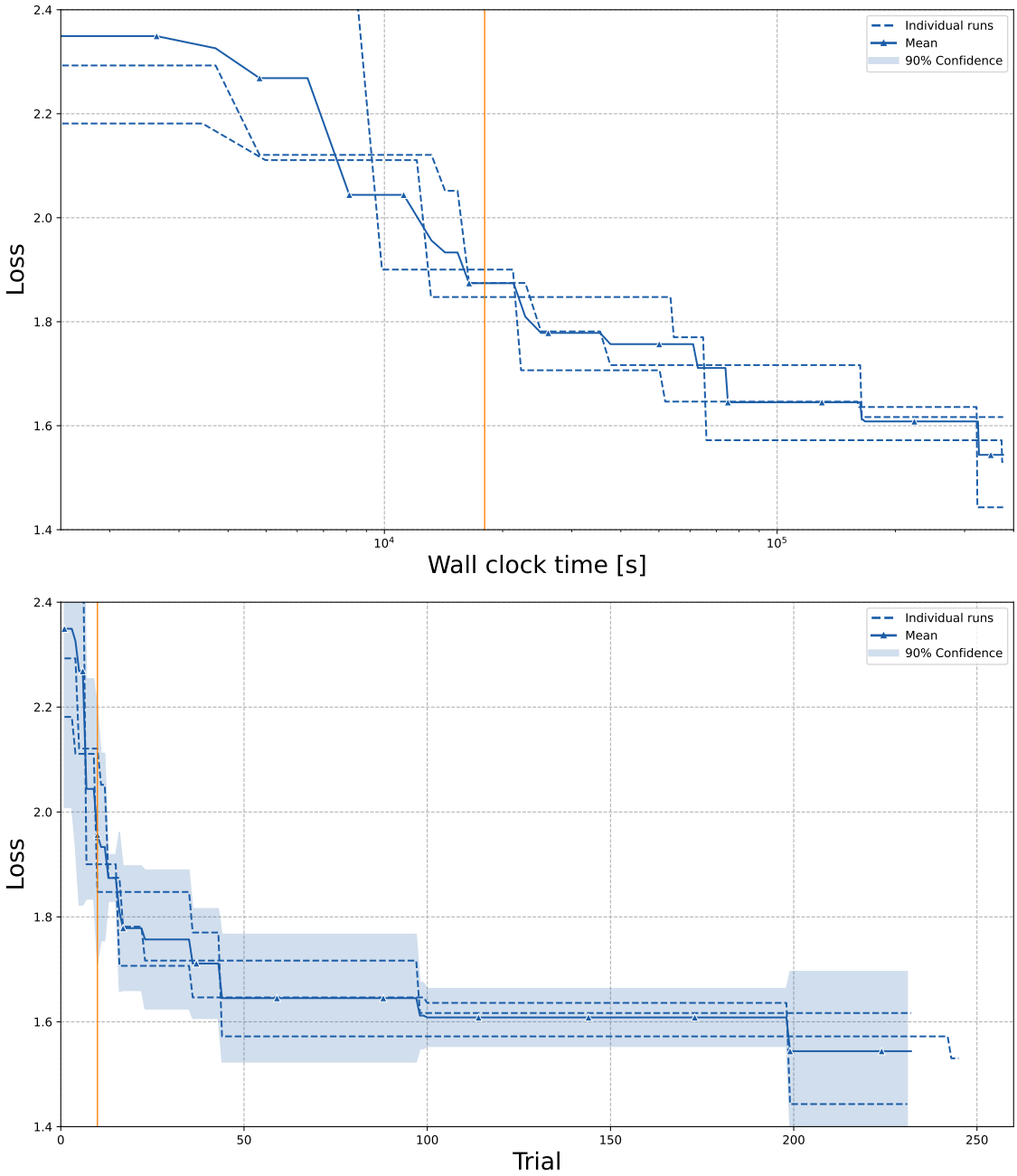


Figure 6.5: Cumulative minimum for BO. The orange vertical line marks where BO and BO-MSR stopped sampling randomly. The confidence interval stretches out from the mean and is only shown where all three runs were still sampling. All three runs evolve quite similar throughout. The largest improvements happen up until around 8×10^4 seconds, or 50 trials, with some minor improvements later on. The confidence interval is relatively small throughout.

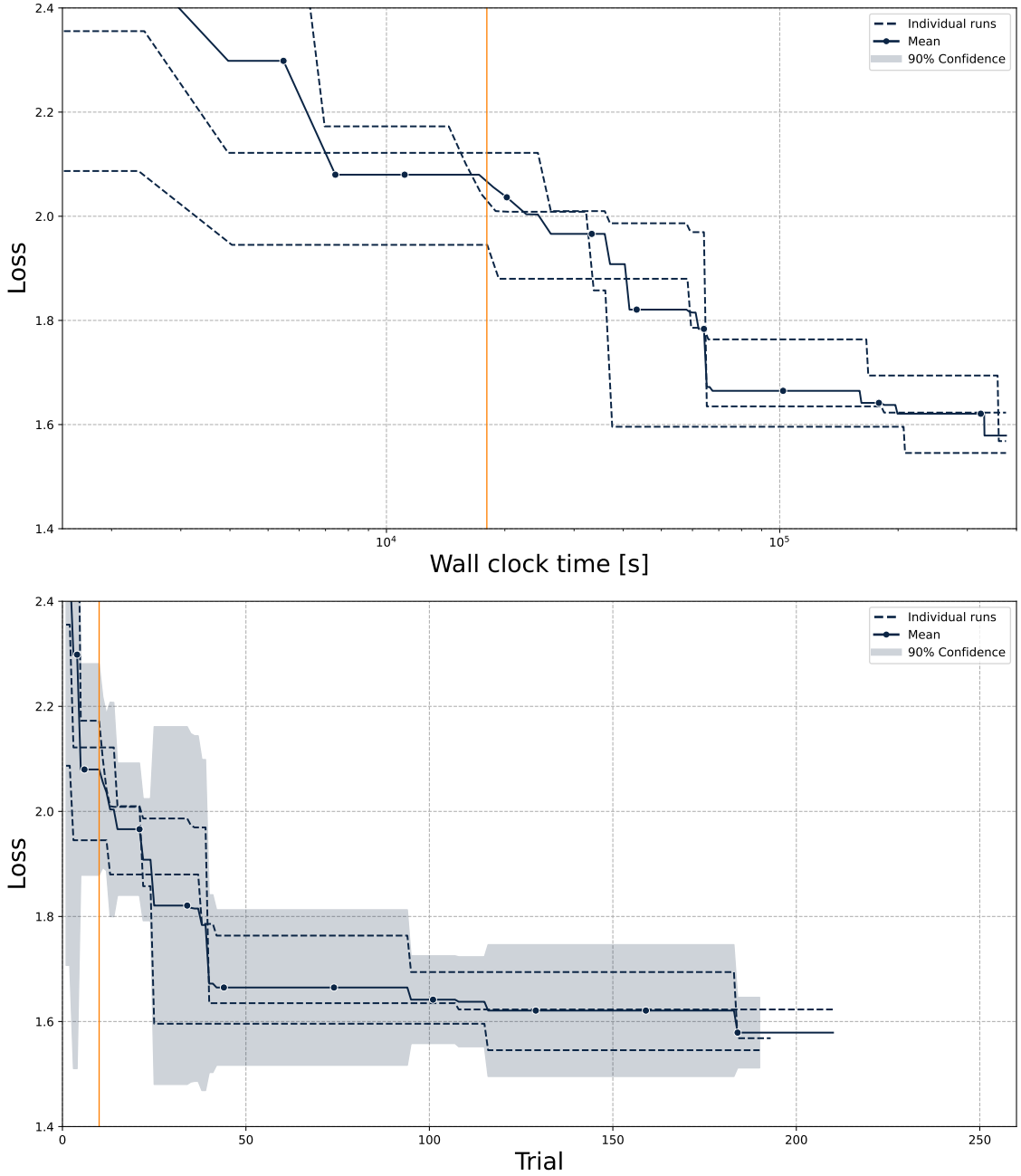


Figure 6.6: Cumulative minimum for BO-MSR. The orange vertical line marks where BO and BO-MSR stopped sampling randomly. The confidence interval stretches out from the mean and is only shown where all three runs were still sampling. All three runs evolve quite similar throughout. The largest improvements happen up until around 8×10^4 seconds, or 50 trials, with some minor improvements later on. The confidence interval is a bit varying up until around 8×10^4 seconds, or 50 trials, and is relatively small after that.

The performance bottleneck made it as mentioned difficult to interpret the results. However, some insight can still be extracted. How many trials stopped at each epoch for RS-MSR and BO-MSR is given in Tables 6.2 and 6.3 respectively, and the distribution of which epoch the trials were stopped at is illustrated in Figure 6.7. The trial number of the first trial stopped in each run of RS-MSR and BO-MSR is given in Table 6.4.

Comparing the individual plots of Figure 6.2, the results show that the trial per wall time ratio affects the anytime and final performance of the methods, as you can see BO-MSR "catching up" with BO some time after BO improves, while they on a trial basis perform similarly. Furthermore, Figures 6.5 and 6.6 show that BO has a slightly smaller confidence interval than BO-MSR for the majority of the time. RS and RS-MSR performed quite similar, both trial per wall time-wise, and regarding the confidence interval.

Tables 6.2 and 6.3 show that RS-MSR stopped a significant amount of more trials than BO-MSR. On average, RS-MSR stopped around 37% of the total trials, or 47% if you subtract the fifty initial trials. BO-MSR only stopped around 14% of the total trials, or 19% if you subtract the fifty initial trials. Tables 6.2 and 6.3 also show that the vast majority of trials were stopped at the earliest epoch possible, as illustrated in Figure 6.7. Table 6.4 further shows that the first trials stopped were stopped quite soon after the required 50 trials had trained on the full budget.

Table 6.2: The table show how many trials were stopped at each epoch for the individual runs of RS-MSR. Most of the trials were stopped at the earliest epoch possible.

Run \ Epoch	2500	2501	2502	2522	2541	2724	2819	4250
1	73	5	1	0	1	0	0	1
2	73	8	0	0	0	1	0	0
3	86	6	0	1	0	0	1	0

Table 6.3: The table show how many trials were stopped at each epoch for the individual runs of BO-MSR. Most of the trials were stopped at the earliest epoch possible.

Run \ Epoch	2500	2501	2505	2506	2516	2523	2565	2614
1	23	2	0	0	0	1	1	0
2	18	3	0	0	0	0	0	0
3	34	4	1	1	1	0	0	1

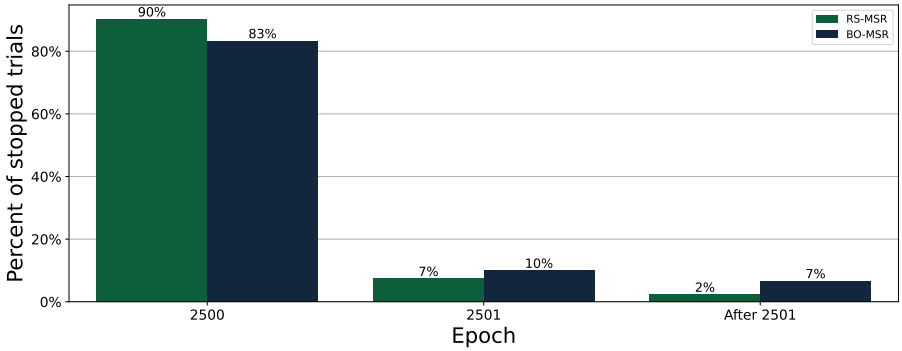


Figure 6.7: Number of early stopped trials for RS-MSR and BO-MSR. The vast majority of terminated trials are terminated as soon as possible, and only a fraction of trials is allowed to continue from the first possible termination point.

Table 6.4: The table shows the earliest stopped trial for the individual runs of RS-MSR and BO-MSR. In all cases, the first stopped trial happened quite quickly after the required 50 trials had been trained to completion.

Run	RS-MSR	BO-MSR
1	52	62
2	52	52
3	51	59

6.3 Robustness and configuration space

The hyperparameter values of the five best configurations are given in Table 6.5, along with the corresponding loss and which method that found the configuration. To investigate the robustness of these configurations, each configuration was run an additional six times (because of the expense budget) after the experiment, yielding a total of 7 runs for each configuration. A mean and standard deviation for the resulting loss of each configuration was calculated from these seven runs and is also given in Table 6.5. The second-best configuration found in the experiment, B2, resulted in both the smallest mean and standard deviation. The largest mean resulted from the fifth-best configuration, B5, and the largest standard deviation came from the best configuration, B1.

The distribution of the loss for each of the five best configurations is given in the form of a box plot in Figure 6.8. The original loss found by B1, B4 and B5 are all marked as outliers. B2 is the only configuration in which all 7 runs managed to improve from $\bar{v}$.

Interestingly, the top five configurations were all less complex than the standard NC. Investigating this a bit further by plotting the distribution of shallower or not shallower configurations in the top 1, 5 and 15 percentile of all trials, one can see in Figure 6.9 that almost two thirds of the configurations were shallower, and therefore

Table 6.5: The table shows the five best configurations found in the experiment, their corresponding loss and which method found them. Each configuration was run an additional six times to calculate a mean and standard deviation for the loss.

ID	Loss	Mean	STD	Learning rate	κ_{shared}	$\kappa_{specific}$	Context	Depth	Method
B1	1.44	2.13	0.36	2.020E-03	8.396E-07	5.061E-09	8	4	BO
B2	1.52	1.98	0.28	2.122E-04	4.465E-08	2.864E-08	2	2	RS
B3	1.53	2.03	0.31	6.251E-04	2.141E-06	3.631E-05	25	2	RS
B4	1.53	2.08	0.31	2.536E-03	8.466E-05	1.596E-10	1	3	RS-MSR
B5	1.53	2.18	0.32	5.155E-03	2.482E-08	6.372E-05	2	3	BO

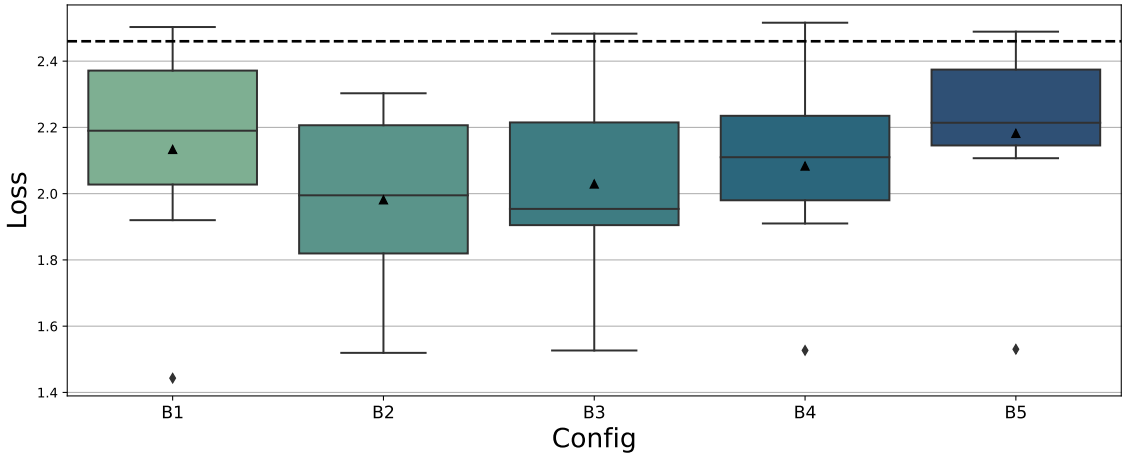


Figure 6.8: Box plot of the result of running the five best configurations an additional six times. The median is shown with a grey horizontal line, and the mean is shown as a black triangle. The grey diamond marks outliers, and the whiskers extend to 1.5× IQR. The black dashed line shows $\bar{v}$. B2 has the lowest mean, while B3 has the lowest median. B2 is the only configuration in which all 7 runs managed to improve from $\bar{v}$. The original loss found by B1, B4 and B5 are all marked as outliers.

also less complex, than the standard NC in the percentiles. The figure also shows that around half of all trials were shallower than the standard NC.

Continuing to investigate this discovery, one can see in Figure 6.10 the distribution of depth in the top 15 percentile. The figure shows a clear trend between depth and number of configurations in the percentile: The shallower the net, the more configurations.

Scatter plots of hyperparameter values and corresponding loss for all trials are given in Figure 6.11, per method in Figures 6.12 to 6.15, and per run per method in Figures C.1 to C.4. Figure 6.11 show that only the learning rate seems to have a well-defined region of well-performing values. The region is fairly consistent per method and for each run as well. The learning rate of B1 to B4 is around, or slightly to the right of, the centre of this region. The learning rate of B5 seems to be closer to the upper limit of the region. Comparing Figures 6.12 and 6.13 to Figures 6.14 and 6.15, it is also clear that the model-based methods sampled more hyperparameter configurations within the well-defined region.

Figure 6.14 shows that BO in addition also favoured larger values of $\kappa_{specifc}$ and

slightly favoured smaller values of κ_{shared}. Figure 6.15 shows that BO-MSR favoured either small or large values for $\kappa_{specifc}$. Figures C.3 and C.4 also illustrates which regions the multivariate TPE favoured in the individual runs for BO and BO-MSR, respectively.

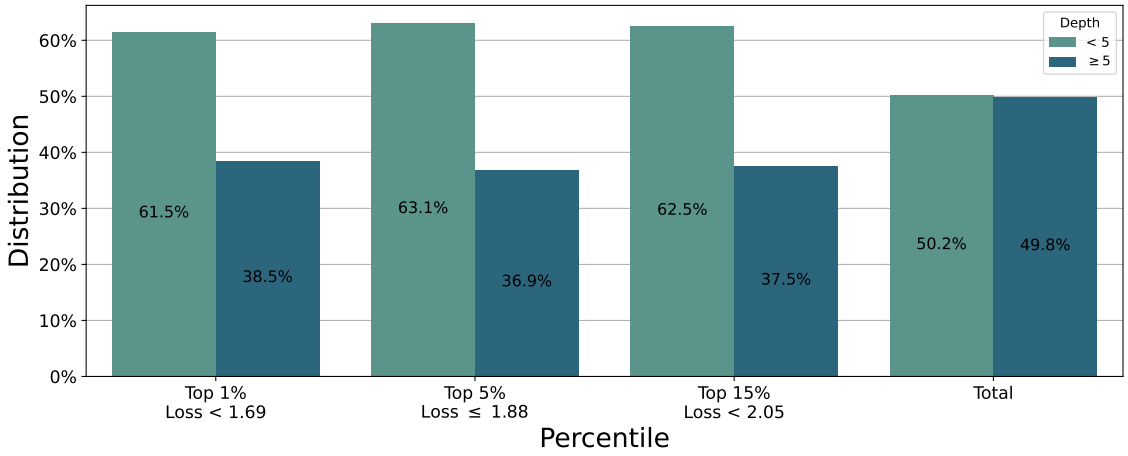


Figure 6.9: Sea green shows the distribution of configurations that were shallower (< 5), and petroleum shows the distribution of configurations that were deeper or equal (≥ 5), than the standard NC, in the top 1, 5 and 15 percentile. The distribution for the total, 2588 trials is also shown. Half of all trials were shallower, while around two-thirds of the trials in the top 1, 5 and 15 percentile were shallower.

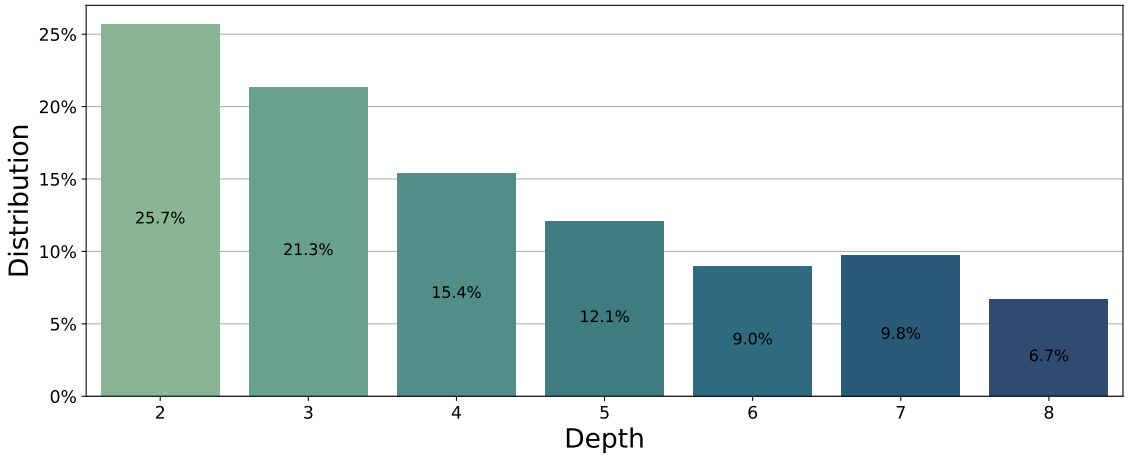


Figure 6.10: Distribution of depth in the top 15 percentile. The plot shows a clear trend: The shallower the net, the more configurations.

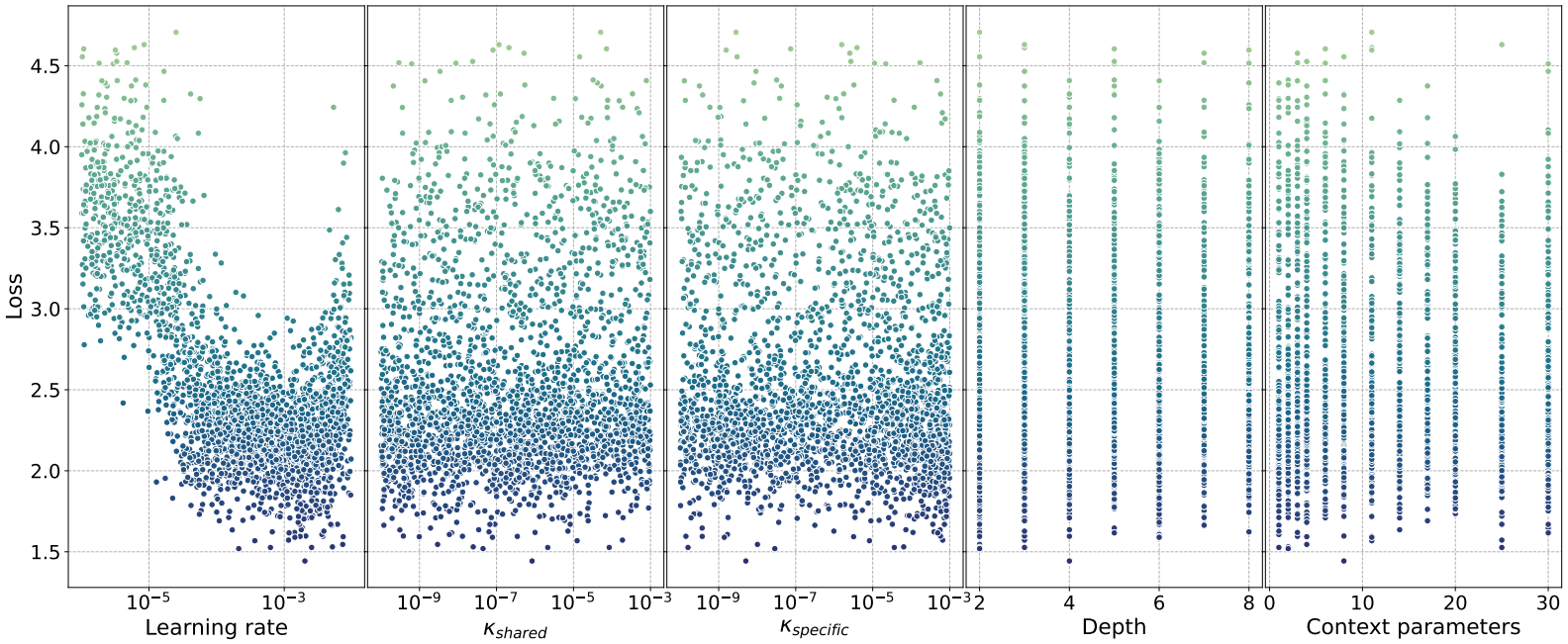


Figure 6.11: Scatter plot of hyperparameters from all trials and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values.

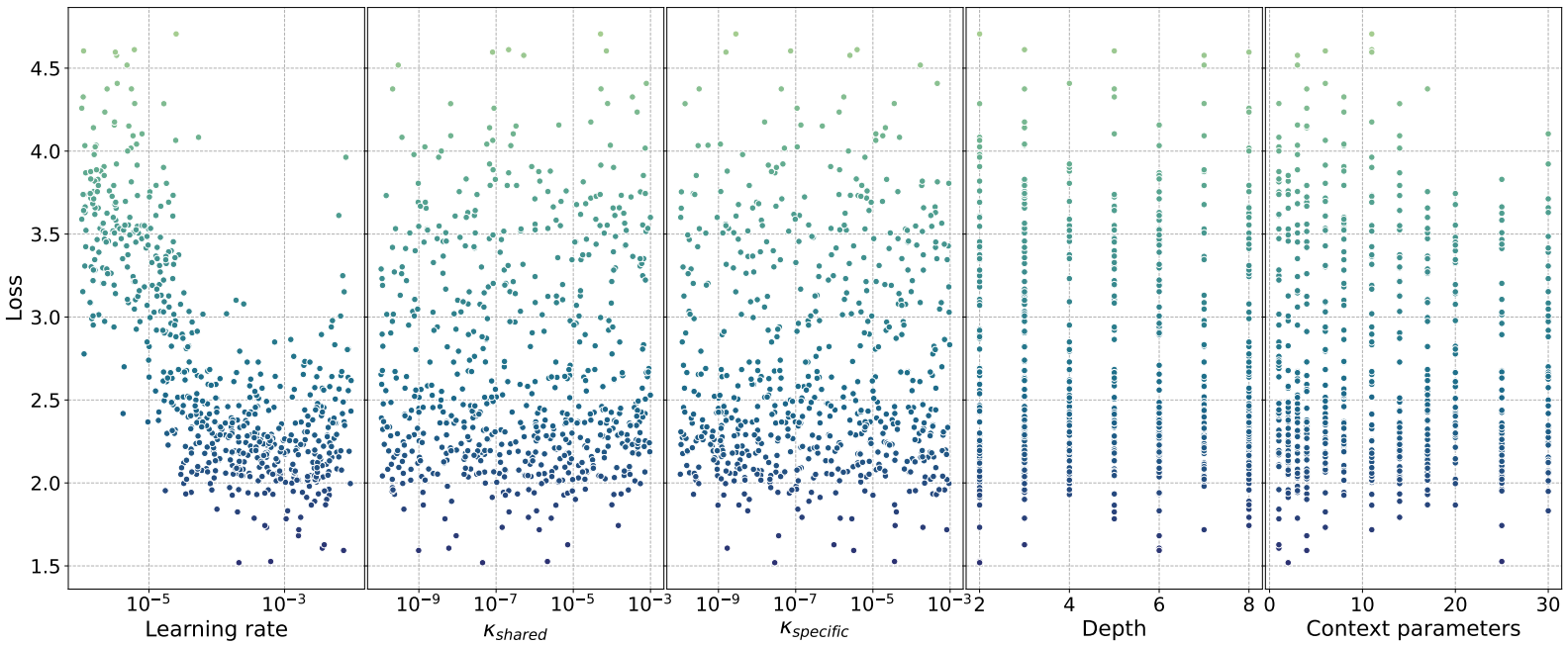


Figure 6.12: Scatter plot of hyperparameters from all trials of RS and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values.

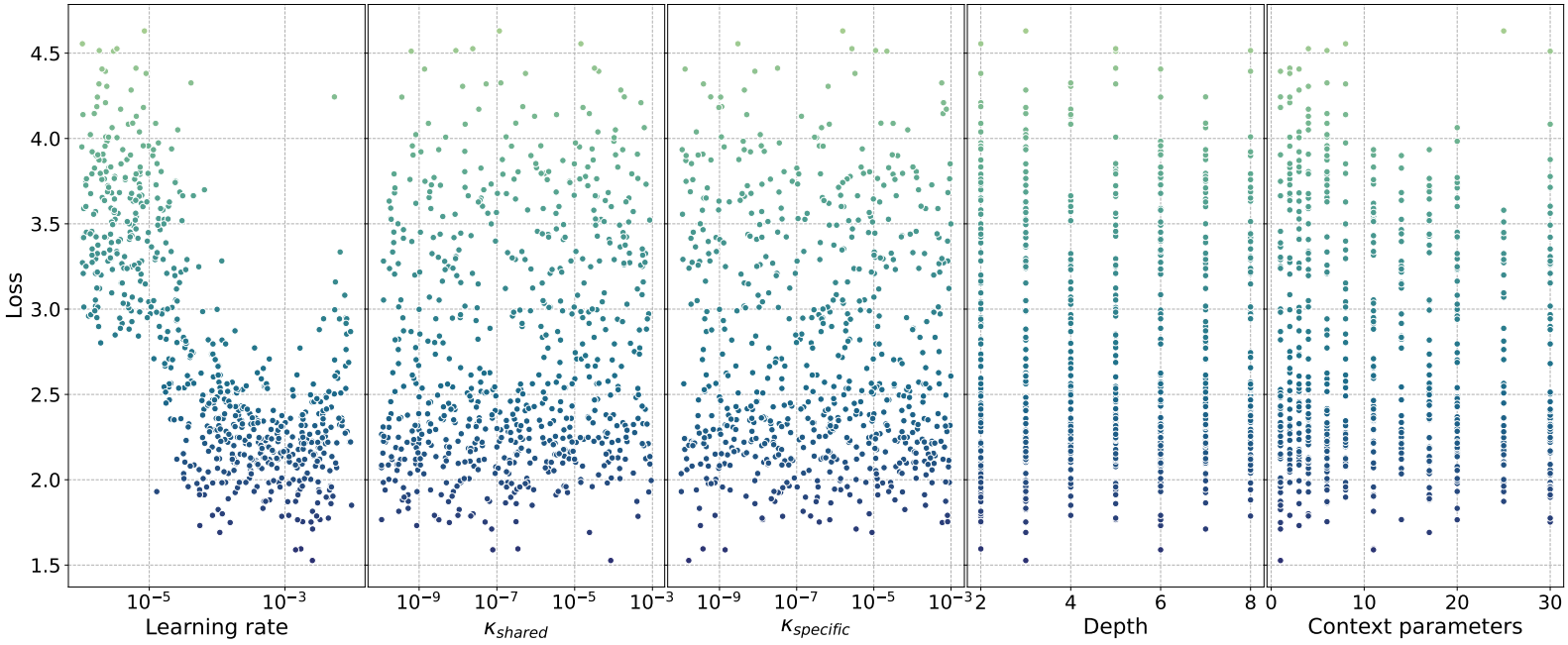


Figure 6.13: Scatter plot of hyperparameters from all trials of RS-MSR and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values.

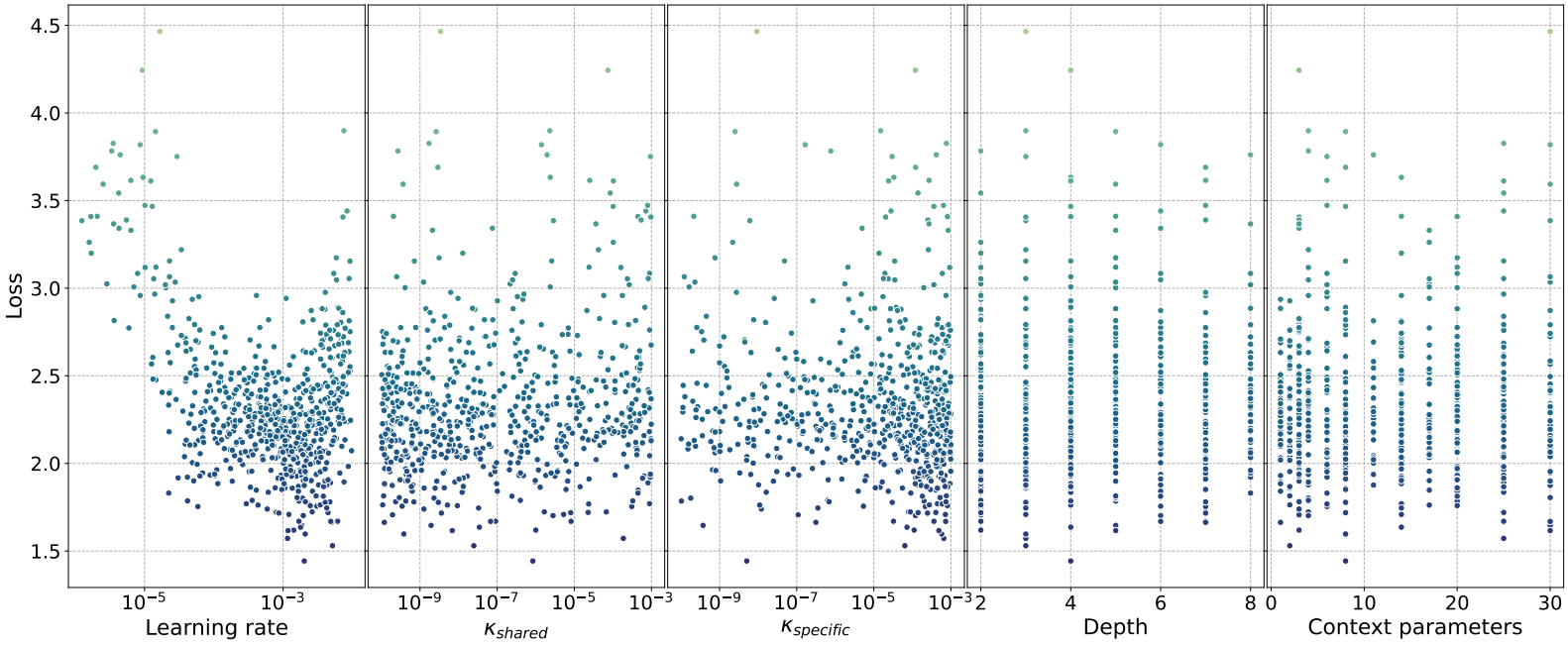


Figure 6.14: Scatter plot of hyperparameters from all trials of BO and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. The learning rate seems to have a clear region of good and bad performing values, and the acquisition function acquired many $\kappa_{specifc}$ with higher values, and slightly more κ_{shared} with smaller values.

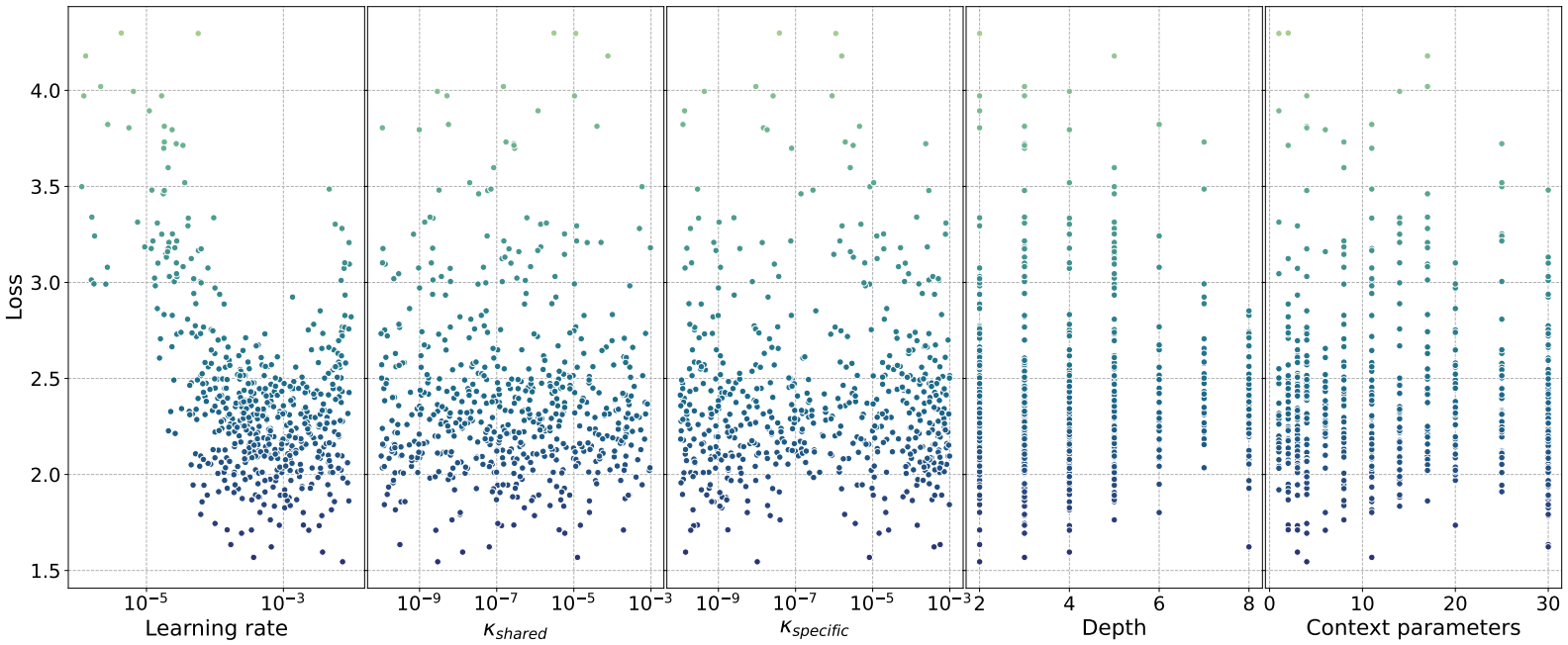


Figure 6.15: Scatter plot of hyperparameters from all trials of BO-MSR and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values, and the acquisition function seemed to somewhat favour either small or large values of $\kappa_{specific}$.

7 | Discussion

This chapter discusses the demonstrated results in Chapter 6. The discussion is structured after the hypothesis defined in Section 5.3.1, before the robustness and the configuration space is discussed in the end.

7.1 H1

The results support H1, as they show that all methods were able to improve from $\bar{v}$. This was just as expected, since no extensive HPO had yet been performed on the NC. The results also showed how quickly each run of each method was able to improve from $\bar{v}$. Running a single experiment overnight for 15 hours, or 5.4×10^4 seconds, while you're away from work between 17:00 and 08:00, showed significant improvement. A simple random search could thus have already improved the performance of the NC without much effort. This underlines the effectiveness of automated HPO.

7.2 H2

Regarding H2, it is difficult to either confirm or reject the hypothesis because of the bottleneck. However, the idea behind using early stopping is to be able to sample more trials per wall time. The results also showed that this ratio is key.

Looking first at the effect of adding early stopping to random search; RS vs. RS-MSR. The results of Table 6.2 show that RS-MSR stopped a significant amount of trials. On average, it stopped around 37% of the total trials, or 47% if you subtract the fifty initial trials. By stopping this significant amount of trials, RS-MSR was able to make up for the performance bottleneck and was in the end able to sample a few more trials than RS. The results show that RS-MSR obtained a lower mean and smaller confidence interval towards the end. This partly supports H2, since lower mean and smaller CI towards the end speaks for a better final performance for RS-MSR.

H2 is furthermore supported by the percentile distribution shown in Figure 6.1. We attribute the fact that RS found more configurations in the top 1% to the fact that this percentile only contains 26 samples. RS-MSR outperformed RS in the top 5 and 15%, which indicates that RS-MSR was able to terminate bad-performing trials early, which allowed for sampling of more, well-performing, trials. Terminating bad-performing trials early means that the algorithm can find well-performing trials

earlier, which again speaks for a better anytime performance for RS.

Looking then at the effect of adding early stopping to BO-TPE by comparing BO and BO-MSR. The cumulative minimum over time in Figure 6.2 seem to show that BO-MSR is catching up with BO some time after BO improves. By looking at the cumulative minimum per trial in the same figure, the results indicate that BO and BO-MSR are performing similarly. This indicates that the early stopping did not affect the KDE in any negative way, and that BO-MSR was able to make as smart acquisitions as BO with partially early terminated samples.

Furthermore, the results of Table 6.3 show that BO-MSR did not stop close to near as many trials as RS-MSR. In fact, BO-MSR only stopped around 14% of the total trials, or 19% if you subtract the fifty initial trials. As such, BO-MSR was not able to make up for the performance bottleneck like RS-MSR was. The similar performance per trial, and the catching up phenomenon, however, indicates that H2 is true. Based on this, and that the early stopping did not seem to negatively affect the KDE, we believe that – if the performance bottleneck had not been present – BO-MSR would have outperformed BO.

We attribute the differences in the distribution of configurations from each method in the top 5, 10, and 15 percentiles (Figure 6.1) to two reasons; 1) BO sampled a significant amount of more configuration than BO-MSR, and thus already had a larger share of the pool, and 2) By having more samples, BO also had more information to base its acquisitions on.

7.2.1 Early stopping aspects

Disregarding the performance bottleneck, there are several aspects that affected the early stopping, and it is difficult to estimate their influence on the final results.

First, what would have happened if the number of minimum epochs required before allowing termination was different? The results showed that the vast majority of the early-stopped trials were terminated as early as possible, and if not, very soon after. Could the required number of epochs then have been lowered, and in that case, how would that affect the TPE? One possible outcome is that by stopping too early, the samples used to build the KDE could be too misleading, which would negatively affect the acquisition.

On the other hand, what would have happened if the required minimum number of epochs was higher? Just as in Aesop's *The Tortoise and the Hare*, maybe some early stopped trials would have found the best final performance if they had been allowed to run to the finish line? This is especially interesting considering the learning rate scheduler kicking in after 4000 epochs, and that the early stopped trials never reached this part of the algorithm. For the model-free methods, it is difficult to say anything about this, as all configurations were sampled at random. For the model-based, however, we assume that as the optimization goes on, the surrogate model should be able to find some well-performing regions, and are searching for the local minima of these regions. Since BO and BO-MSR performed similarly per trial, we

believe that we at least did not stop the trials too early. Nevertheless, this does not exclude the possibility of a tortoise among many hares.

Secondly, the number of minimum required trials could have been changed. Table 6.4 showed that the first terminated trials were stopped soon after the minimum required samples. By lowering the minimum required number of samples, would the median still have been representative in the early stages of the optimization? If we assume that the loss $v = f(\boldsymbol{\lambda})$ is normally distributed, reducing the minimum number of required trials by, e.g., 50% would then not have resulted in that much a larger confidence interval. The minimum required samples could thus probably have been reduced, at least a bit. On the other hand, reducing the minimum number of required samples could have affected the TPE, as possibly even fewer samples could have been trained to completion.

In contrast, it could also have been possible to increase the minimum number of required samples. However, considering how early the first trials were stopped, we believe that the minimum of required samples was at least not too small. This is also supported by the fact that BO and BO-MSR performed similarly per trial, and as such increasing the minimum number of required samples would not have resulted in significant improvements for the KDE. Interactions between the minimum required epochs, and the minimum required samples could also have been considered – but that would result in too many what-ifs.

A final factor affecting the early stopping is the calculation of the loss (Equation 5.1). Does using the average over the last ten epochs result in a too smooth loss? Considering that at time t, the MSR 1) Uses the best-observed loss up until t for the current trial, and 2) The median is obtained from the running average up until time t of the completed trials, we believe no. We believe so because 1) If the loss of a trial had a sudden fall in t, and in t_{+1} jumped back up to where it was in t_{-1}, then that sudden fall would have affected the stopping rule too much, as the best-observed loss up until that point in time is used, and 2) the running average at of $[0, t]$ is smoother than the average of $[t_{-10}, t]$, for $t > 10$, and much smoother for $t >> 10$.

7.3 H3

The results of Table 6.1 showed that the final mean of BO was lower than the mean of RS, and Figures 6.2, 6.3 and 6.5 shows that the mean of RS and BO develop similar throughout the experiment, but the confidence interval of BO is significantly smaller than that of RS. The results, therefore, indicate that BO has better anytime and final performance than RS.

The same can be said about BO-MSR vs RS-MSR, even though RS-MSR achieved the most trials per wall time, and BO-MSR the fewest. The results of Table 6.1 showed that the final mean of BO-MSR was lower than the mean of RS-MSR, and Figures 6.2, 6.4 and 6.6 shows that the mean of RS-MSR and BO-MSR develop similar throughout the experiment, but the confidence interval of BO-MSR is significantly smaller than that of RS-MSR. The results, therefore, indicate that BO-MSR

has better anytime and final performance than RS-MSR. Both these statements support H3.

H3 is further supported by the percentile plots of Figure 6.1. BO and BO-MSR consistently outperform RS and RS-MSR, respectively, in all shown percentiles. This means that the model-based methods are able to find more well-performing hyperparameter configurations than the model-free methods, suggesting both a stronger anytime and final performance.

However, the development over time between BO and RS (we only look at these configurations to be able to disregard the performance bottleneck), was more similar than expected. This prompts the question; Was the surrogate able to model the response surface well enough?

One aspect that certainly affects the surrogate's modelling ability, is the size of the configuration space to the allowed time ratio. It is difficult to say anything about this from the results in this particular case. However, increasing the ratio, by either decreasing the size of the configuration space or increasing the allowed time, results in either samples that better cover the configuration space, or more samples. Either way, this should allow the surrogate to better model the response surface.

The scatter plot of Figure 6.11 shows that only the learning rate seems to have a well-defined region of well-performing values. That might have difficultated the response surface modelling. This concept is illustrated in Figure 7.1. The plot shows the division into the two categories, ℓ and g, for the learning rate, κ_{shared}, and $\kappa_{specific}$ from all 2588 configurations of the experiment, along with a univariate KDE for the three hyperparameters. Compared to Figure 3.12, the KDE of Figure 7.1 does not provide much clarity about good and bad hyperparameter values.

The multivariate TPE was used in the experiment, but the point still stands. Imagine, if possible, that both the scatter and KDE-plot is a six-dimensional hypercube, where five of the dimensions are the hyperparameters, and the last dimension is either the loss or the density estimation. If the density estimate then looks anything close to what, e.g., the density estimate of $\kappa_{specific}$ in Figure 7.1, then it makes sense that the acquisition function was not able to locate any well-performing local minima.

On the other hand, what would have happened if the univariate TPE had been used? Look at the learning rate in Figure 7.1 for example. The univariate TPE might have been able to optimize this parameter better, since it uses one KDE for each hyperparameter. Then again, with no covariance modelled, it might also have been more difficult for the univariate TPE to model the other hyperparameters.

7.4 Robustness and configuration space

The results in Table 6.5 and Figure 6.8 indicates that none of the five best configurations were robust. The best loss found by B1, B4 and B5 in the original experiment, all fell outside 1.5 × the IQR, while the loss found by B2 and B3 in the original

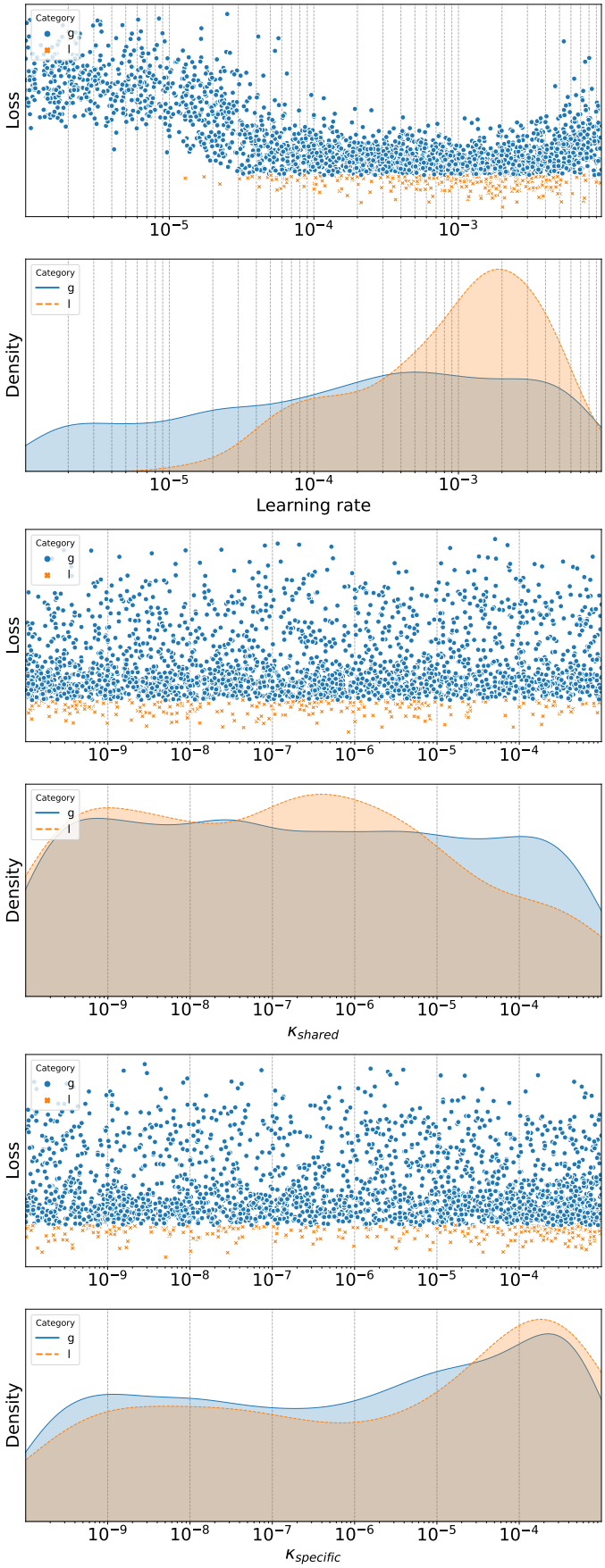


Figure 7.1: Scatter plot of the learning rate, κ_{shared}, and $\kappa_{specific}$ from all 2588 configurations of the experiment, equal to Figure 6.11. The samples have been split into two categories, ℓ and g. Below the scatter plot is a univariate KDE for each of the categories.

experiment is at the endpoint of the bottom whiskers. The mean for each configuration was also found to be significantly higher than the loss found in the experiment. Considering the small values, the standard deviation in each configuration was also quite high.

B1, B4 and B5 are arguably the least robust configurations, because of their high mean, median and standard deviation. B2 and B3 are more similar, but B3 has a higher mean and STD, in addition to an observation above $\bar{v}$, which renders it less robust. That leaves B2 as the most robust hyperparameter configuration, out of the top five configurations.

Table 6.5 showed that the top five configurations were all less complex than the standard NC. In addition, Figure 6.9 showed that around two thirds of the configurations in the top 1, 5 and 15 percentile were shallower than the NC. Figure 6.10 furthermore showed that, in this case study, the shallower the net, the more configurations in the well-performing percentile. This is somewhat contradictory to what was expected, since practise have shown that large models that have been regularized appropriately tends to work best. However, what defines large, and what defines appropriately, are of course case specific.

These results might therefore indicate that the allowed regularization in this experiment was not appropriate enough for the deeper models. They might also indicate that the standard NC was too complex. Finally, they might also just underline the fact that deeper models are more difficult to train, even with residual blocks, explained as the degradation problem in Section 3.2.5.

Nevertheless, the shallowness might have affected the robustness of the top five configurations, as they, in addition to being shallow, also used quite weak regularization. The shallow weak-regularized networks might be more prone to stochasticities in the learning algorithm, as they have fewer and less regularized learning parameters. Thus, deeper and heavier regularized networks might be more robust.

Considering the scatter plots of Figures 6.11 to 6.15, 7.1 and C.1 to C.4, a natural question to ask is if the configuration space could have been further limited. Considering the well-defined region of the learning rate, increasing the lower bound to around 10^{-5} could probably have been done without affecting the performance of the methods negatively.

Regarding the other hyperparameters, it is difficult to say. One thing that could have been done with the regularization parameters, κ_{shared} and $\kappa_{specific}$, was to allow for either strong or weak regularization, by splitting their range into e.g: $\{[10^{-10}, 10^{-7}], [10^{-5}, 10^{-2}]\}$. This is sort of contradictory to the out-take from Goodfellow et al. (2016) in Section 5.1 which stated 'that the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately'. Beforehand, it was uncertain what appropriately would mean, but considering that neither BO nor BO-MSR either favoured values of κ_{shared} and $\kappa_{specific}$ in the middle of our defined range, and that no general well-performing region could be identified, we could probably have decreased the size of the configuration space by splitting the range into a weak and strong part. In addition, if a majority of the well-performing configurations had κ_{shared} and $\kappa_{specific}$ close to the

middle, another, smaller, experiment with only the middle range could have been run after.

Another way of decreasing the configuration space could also have been to use fewer hyperparameters. Given that only the learning rate seems to have a well-defined region of well-performing hyperparameters, an interesting possibility would have been to only optimize this hyperparameter and keep everything else fixed. On one hand, this could lead to worse single-experiment results, as we exclude many possible configurations, but on the other hand it might lead to more robust configurations. With a smaller configuration space, the surrogate should be able to model the response of the objective function better, which should increase the robustness effect described in Section 3.3.6.

Another aspect with the configuration space is the hyperparameters that were chosen to be optimized, or rather the ones that were not chosen. Interesting hyperparameters such as which optimizer to use, the number of epochs, mini-batch size or different activation functions could also have been considered. How increasing the configuration space, or replacing one or more of the chosen hyperparameters with one or more of these hyperparameters, would have affected the optimization is impossible to say anything informative about. However, given that BO was less superior to RS than expected, and that the top five configurations were not that robust, optimizing other parameters is rendered an interesting topic for future work.

8 | Conclusion

The case study demonstrated one way of how one can take special considerations, such as a restricted time and cost budget and key aspects of machine learning algorithms, into account when selecting a hyperparameter optimization algorithm and which hyperparameters to optimize. Furthermore, the results of the case study showed that:

- All identified HPO-algorithms found solutions that were significantly better than the benchmark $\bar{v}$. They all also quickly improved from the benchmark.
- Out of the identified HPO-algorithms, BO performed the best. However, based on the results, we believe BO with the median stopping rule would have performed best if the performance bottleneck had not been present.
- None of the top five configurations were particularly robust, but out of the five, B2 was the most robust.
- The results are generally in line with the theory, and mostly support the hypothesis defined in Section 5.3.1.

8.1 Suggestions for future work

8.1.1 Different HPO-algorithms

BO did not outperform RS as much as expected, which might have been because of difficulties with modelling the response surface. Other SMBO methods, such as SMAC or BO with the univariate TPE, could therefore be more suited to this case than BO with multivariate TPE. Difficulties with modelling the response surface might also speak in favour of model-free methods, such as the population-based methods. Appropriately selecting the HPO-algorithm is as described paramount for achieving satisfying results. Although the selection of BO and BO-MSR were justified, other methods might be more suitable. Since the evaluation of HPO-algorithms is mostly based on empirical results, experimenting with different HPO-algorithms should be investigated in future works.

8.1.2 Appropriate regularization

The top five configurations were not particularly robust, and most of the top-performing configurations were shallower than the standard NC. One reason for

that might have been that we did not allow for appropriate regularization of the deeper models. Practice have shown that large and properly regularized networks tends to perform better than shallower networks. Therefore, experimenting with other means of regularization should be investigated in future works, to possibly improve anytime and final performance.

8.1.3 Configuration space

The similar performance of BO and RS, and lack of robustness in the top five configurations, could also be because of a poorly selected configuration space. Limiting the configuration space is key for the model-based methods to effectively acquire well-performing configurations. Therefore, experimenting with smaller ranges and optimizing fewer hyperparameters should be investigated. Furthermore, many interesting hyperparameters were not optimized in this work. With only the learning rate having a well-defined region of well-performing values, other hyperparameters might have been better to optimize. It should therefore also be experimented with optimizing other hyperparameters.

8.1.4 Early stopping

The early stopping did not seem to affect the KDE negatively, and (disregarding the bottleneck) early stopping more aggressively could therefore yield greater speed-up. Therefore, it should be experimented with requiring fewer full-budget trials, and/or fewer epochs. It should also be investigated if other multi-fidelity techniques, that are feasible to use within the expense budget, are more appropriate than the median stopping rule.

Bibliography

Akiba, Takuya et al. (2019). 'Optuna: A Next-generation Hyperparameter Optimization Framework'. In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Bergstra, James, Rémi Bardenet et al. (2011). 'Algorithms for hyper-parameter optimization'. In: *Advances in neural information processing systems* 24.

Bergstra, James and Yoshua Bengio (2012). 'Random search for hyper-parameter optimization.' In: *Journal of machine learning research* 13.2.

Bergstra, James, Daniel Yamins and David Cox (June 2013). 'Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures'. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 1. Atlanta, Georgia, USA: PMLR, pp. 115–123. URL: https://proceedings.mlr.press/v28/bergstra13.html.

Breiman, Leo (2001). 'Random forests'. In: *Machine learning* 45.1, pp. 5–32.

Brochu, Eric, Vlad M Cora and Nando De Freitas (2010). 'A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning'. In: *arXiv preprint arXiv:1012.2599*.

Caruana, Rich (1997). 'Multitask learning'. In: *Machine learning* 28.1, pp. 41–75.

Dan, Simon (2013). *Evolutionary Optimization Algorithms*. Wiley. ISBN: 9780470937419. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=597827&site=ehost-live.

Domhan, Tobias, Jost Tobias Springenberg and Frank Hutter (2015). 'Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves'. In: *Twenty-fourth international joint conference on artificial intelligence*.

Elshawi, Radwa, Mohamed Maher and Sherif Sakr (2019). *Automated Machine Learning: State-of-The-Art and Open Challenges*. arXiv: 1906.02287 [cs.LG].

Elsken, Thomas, Jan Hendrik Metzen and Frank Hutter (2019). 'Neural architecture search: A survey'. In: *The Journal of Machine Learning Research* 20.1, pp. 1997–2017.

Falkner, Stefan, Aaron Klein and Frank Hutter (July 2018a). 'BOHB: Robust and Efficient Hyperparameter Optimization at Scale'. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 1437–1446. URL: https://proceedings.mlr.press/v80/falkner18a.html.

Falkner, Stefan, Aaron Klein and Frank Hutter (July 2018b). *Supplementary material for: BOHB: Robust and Efficient Hyperparameter Optimization at Scale*. URL: http://proceedings.mlr.press/v80/falkner18a/falkner18a-supp.pdf.

Feurer, Matthias and Frank Hutter (2019). 'Hyperparameter Optimization'. In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff and Joaquin Vanschoren. Cham: Springer International Publishing, pp. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

Feurer, Matthias, Aaron Klein et al. (2019). 'Auto-sklearn: Efficient and Robust Automated Machine Learning'. In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff and Joaquin Vanschoren. Cham: Springer International Publishing, pp. 113–134. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

Glover, Fred W and Gary A Kochenberger (2003). *Handbook of Metaheuristics*. Vol. 57. Springer Science & Business Media. ISBN: 1-4020-7263-5. URL: https://doi.org/10.1007/b101874.

Goh, Gabriel (2017). 'Why Momentum Really Works'. In: *Distill*. DOI: 10.23915/distill.00006. URL: http://distill.pub/2017/momentum.

Golovin, Daniel et al. (2017). 'Google vizier: A service for black-box optimization'. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1487–1495.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Grimstad, Bjarne (2020). *TTK28-Courseware*. https://github.com/bgrimstad/TTK28-Courseware.

He, Kaiming et al. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. DOI: 10.48550/ARXIV.1502.01852. URL: https://arxiv.org/abs/1502.01852.

— (June 2016). 'Deep Residual Learning for Image Recognition'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Hinton, Geoffrey, Nitish Srivastava and Kevin Swersky (2012). 'Neural networks for machine learning lecture 6a overview of mini-batch gradient descent'. In: *Cited on* 14.8, p. 2.

Hiroyuki, Vincent Yamazaki (2020). *"Multivariate" TPE Makes Optuna Even More Powerful*. https://tech.preferred.jp/en/blog/multivariate-tpe-makes-optuna-even-more-powerful/. [Online; accessed 16-May-2022].

Hutter, Frank, Holger H Hoos and Kevin Leyton-Brown (2011). 'Sequential model-based optimization for general algorithm configuration'. In: *International conference on learning and intelligent optimization*. Springer, pp. 507–523.

Hutter, Frank, Lars Kotthoff and Joaquin Vanschoren (2019). *Automated machine learning: methods, systems, challenges*. Springer Nature.

Jamieson, Kevin and Ameet Talwalkar (May 2016). 'Non-stochastic Best Arm Identification and Hyperparameter Optimization'. In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, pp. 240–248. URL: https://proceedings.mlr.press/v51/jamieson16.html.

Kandasamy, Kirthevasan et al. (Aug. 2017). 'Multi-fidelity Bayesian Optimisation with Continuous Approximations'. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1799–1808. URL: https://proceedings.mlr.press/v70/kandasamy17a.html.

Karnin, Zohar, Tomer Koren and Oren Somekh (2013). 'Almost optimal exploration in multi-armed bandits'. In: *International Conference on Machine Learning*. PMLR, pp. 1238–1246.

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

Kingma, Diederik P and Jimmy Ba (2014). 'Adam: A method for stochastic optimization'. In: *arXiv preprint arXiv:1412.6980*.

Klein, Aaron, Stefan Falkner et al. (2017). 'Fast bayesian hyperparameter optimization on large datasets'. In: *Electronic Journal of Statistics* 11.2, pp. 4945–4968.

Klein, Aaron and Frank Hutter (2019). *Tabular Benchmarks for Joint Architecture and Hyperparameter Optimization*. arXiv: 1905.04970 [cs.LG].

Kohavi, Ron and George H. John (1995). 'Automatic Parameter Selection by Minimizing Estimated Error'. In: *Machine Learning Proceedings 1995*. Ed. by Armand Prieditis and Stuart Russell. San Francisco (CA): Morgan Kaufmann, pp. 304–312. ISBN: 978-1-55860-377-6. DOI: https://doi.org/10.1016/B978-1-55860-377-6.50045-1. URL: https://www.sciencedirect.com/science/article/pii/B9781558603776500451.

Kumar, Siddharth Krishna (2017). *On weight initialization in deep neural networks*. DOI: 10.48550/ARXIV.1704.08863. URL: https://arxiv.org/abs/1704.08863.

Li, Lisha et al. (2017). 'Hyperband: A novel bandit-based approach to hyperparameter optimization'. In: *The Journal of Machine Learning Research* 18.1, pp. 6765–6816.

Liaw, Richard et al. (2018). 'Tune: A Research Platform for Distributed Model Selection and Training'. In: *arXiv preprint arXiv:1807.05118*.

Lu, Zhou et al. (2017). 'The expressive power of neural networks: A view from the width'. In: *Advances in neural information processing systems* 30.

Mahajan, Aditya and Demosthenis Teneketzis (2008). 'Multi-armed bandit problems'. In: *Foundations and applications of sensor management*. Springer, pp. 121–151.

Mendoza, Hector et al. (June 2016). 'Towards Automatically-Tuned Neural Networks'. In: *Proceedings of the Workshop on Automatic Machine Learning*. Ed. by Frank Hutter, Lars Kotthoff and Joaquin Vanschoren. Vol. 64. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, pp. 58–65. URL: https://proceedings.mlr.press/v64/mendoza_towards_2016.html.

Merkel, Dirk (2014). 'Docker: lightweight linux containers for consistent development and deployment'. In: *Linux journal* 2014.239, p. 2.

Nair, Vinod and Geoffrey E Hinton (2010). 'Rectified linear units improve restricted boltzmann machines'. In: *Icml*.

Nwankpa, Chigozie et al. (2018). 'Activation functions: Comparison of trends in practice and research for deep learning'. In: *arXiv preprint arXiv:1811.03378*.

Paszke, Adam et al. (2019). 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Provost, Foster, David Jensen and Tim Oates (1999). 'Efficient progressive sampling'. In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 23–32.

Ruder, Sebastian (2016). 'An overview of gradient descent optimization algorithms'. In: *arXiv preprint arXiv:1609.04747*.

— (2017). 'An overview of multi-task learning in deep neural networks'. In: *arXiv preprint arXiv:1706.05098*.

Sandnes, Anders T., Bjarne Grimstad and Odd Kolbjørnsen (2021). 'Multi-task learning for virtual flow metering'. In: *Knowledge-Based Systems* 232, p. 107458. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2021.107458. URL: https://www.sciencedirect.com/science/article/pii/S0950705121007206.

Shahriari, Bobak et al. (2015). 'Taking the human out of the loop: A review of Bayesian optimization'. In: *Proceedings of the IEEE* 104.1, pp. 148–175.

Snoek, Jasper, Hugo Larochelle and Ryan P Adams (2012). 'Practical bayesian optimization of machine learning algorithms'. In: *Advances in neural information processing systems* 25.

Talbi, El-Ghazali (2009). *Metaheuristics: from design to implementation*. Vol. 74. John Wiley & Sons.

Thorn, R, G A Johansen and B T Hjertaker (Oct. 2012). 'Three-phase flow measurement in the petroleum industry'. In: *Measurement Science and Technology* 24.1, p. 012003. DOI: 10.1088/0957-0233/24/1/012003. URL: https://doi.org/10.1088/0957-0233/24/1/012003.

Thornton, Chris et al. (2013). 'Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms'. In: KDD '13. Chicago, Illinois, USA: Association for Computing Machinery, pp. 847–855. ISBN: 9781450321747. DOI: 10.1145/2487575.2487629. URL: https://doi.org/10.1145/2487575.2487629.

Wikipedia contributors (2022a). *Boolean satisfiability problem — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=1064645537. [Online; accessed 15-February-2022].

— (2022b). *CPLEX — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=CPLEX&oldid=1064098947. [Online; accessed 15-February-2022].

Williams, Christopher K and Carl Edward Rasmussen (2006). *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA.

Yang, Li and Abdallah Shami (2020). 'On hyperparameter optimization of machine learning algorithms: Theory and practice'. In: *Neurocomputing* 415, pp. 295–316.

Ying, Chris et al. (June 2019). 'NAS-Bench-101: Towards Reproducible Neural Architecture Search'. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 7105–7114. URL: https://proceedings.mlr.press/v97/ying19a.html.

Appendix

A Google Compute Engine setup

Series:	N1
Machine type:	n1-standard-4
CPU platform:	Automatic
GPU type:	NVIDIA Tesla K80
Disk image:	c2-deeplearning-pytorch-1-11-cu113-v20220316-debian-10
Persistent memory:	100 GB

B Python implementation of the experiment

```
from ray import tune
from ray.tune.suggest.basic_variant import BasicVariantGenerator

# max_concurrent=1 ensures a completely sequential run, i.e. only one trial is run at a time
search_alg = BasicVariantGenerator(
    constant_grid_search=False,
    points_to_evaluate=initial_configs,
    max_concurrent=1
)

tune.run(
    search_alg=search_alg,
)
```

Listing 1: Python implementation of RS.

```python
from ray import tune
from ray.tune.schedulers import MedianStoppingRule
from ray.tune.suggest.basic_variant import BasicVariantGenerator

scheduler = MedianStoppingRule(
    time_attr="training_iterations",
    grace_period=2500,
    min_samples_required=50,
)

# max_concurrent=1 ensures a completely sequential run, i.e. only one trial is run at a time
search_alg = BasicVariantGenerator(
    constant_grid_search=False,
    points_to_evaluate=initial_configs,
    max_concurrent=1
)

tune.run(
    search_alg=search_alg,
    scheduler=scheduler,
)
```

Listing 2: Python implementation of RS-MSR.

```python
from optuna.samplers import TPESampler
from ray import tune
from ray.tune.suggest.optuna import OptunaSearch

sampler = TPESampler(
    multivariate=True,
)

search_alg = OptunaSearch(
    sampler=sampler,
    points_to_evaluate=initial_configs
)

# This ensures a completely sequential run, i.e. only one trial is run at a time
search_alg = tune.suggest.ConcurrencyLimiter(
    search_alg,
    max_concurrent=1
)

tune.run(
    search_alg=search_alg,
)
```

Listing 3: Python implementation of BO.

```python
from optuna.samplers import TPESampler
from ray import tune
from ray.tune.schedulers import MedianStoppingRule
from ray.tune.suggest.optuna import OptunaSearch

sampler = TPESampler(
    multivariate=True,
)

search_alg = OptunaSearch(
    sampler=sampler,
    points_to_evaluate=initial_configs
)

# This ensures a completely sequential run, i.e. only one trial is run at a time
search_alg = tune.suggest.ConcurrencyLimiter(
    search_alg,
    max_concurrent=1
)

scheduler = MedianStoppingRule(
    time_attr="training_iterations,
    grace_period=2500,
    min_samples_required=50,
)

tune.run(
    search_alg=search_alg,
    scheduler=scheduler,
)
```

Listing 4: Python implementation of BO-MSR.

C Hyperparameter scatter plots

Continues on next page due to large figure size.

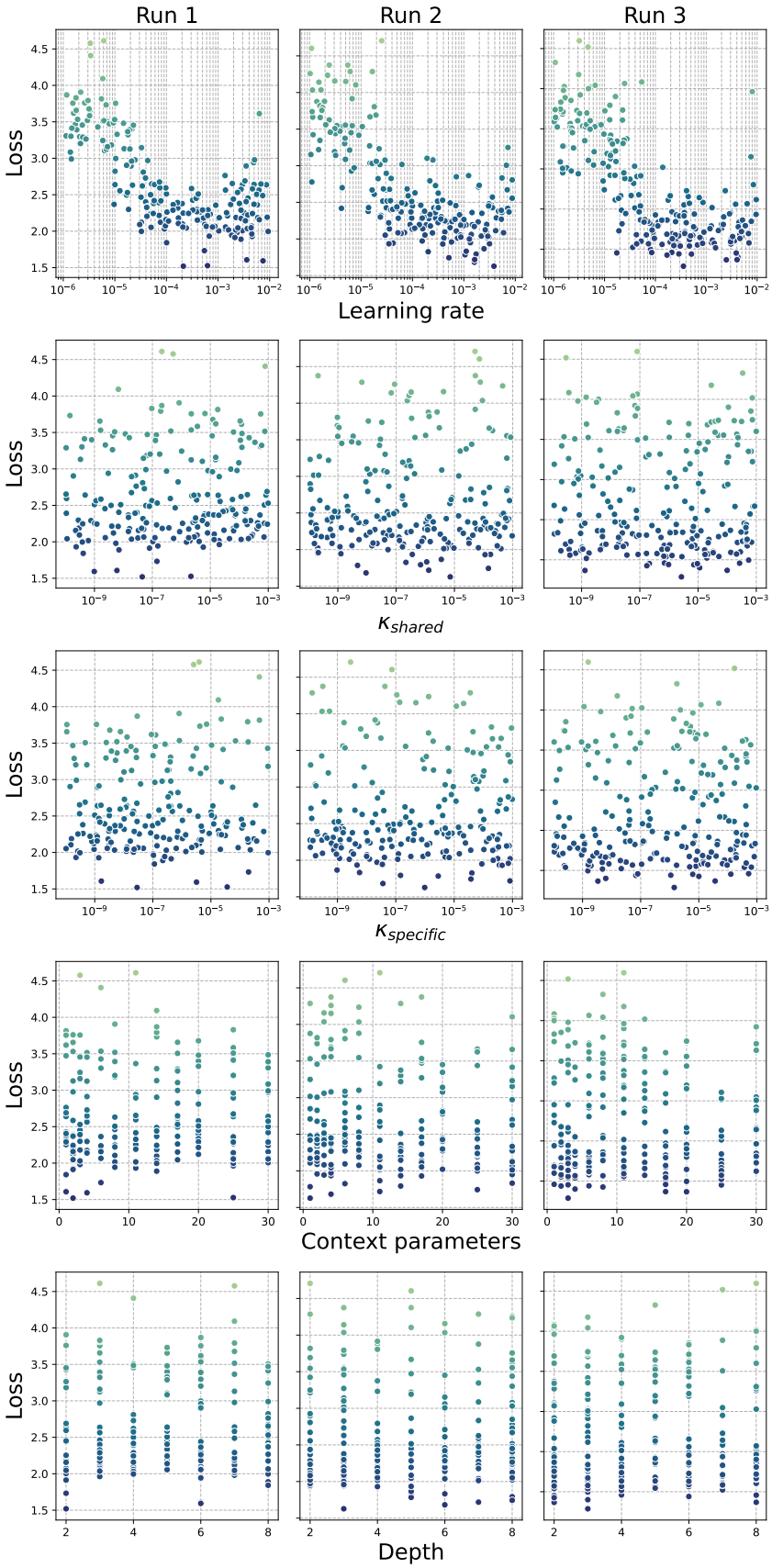


Figure C.1: Scatter plot of hyperparameters from individual runs of RS and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values.

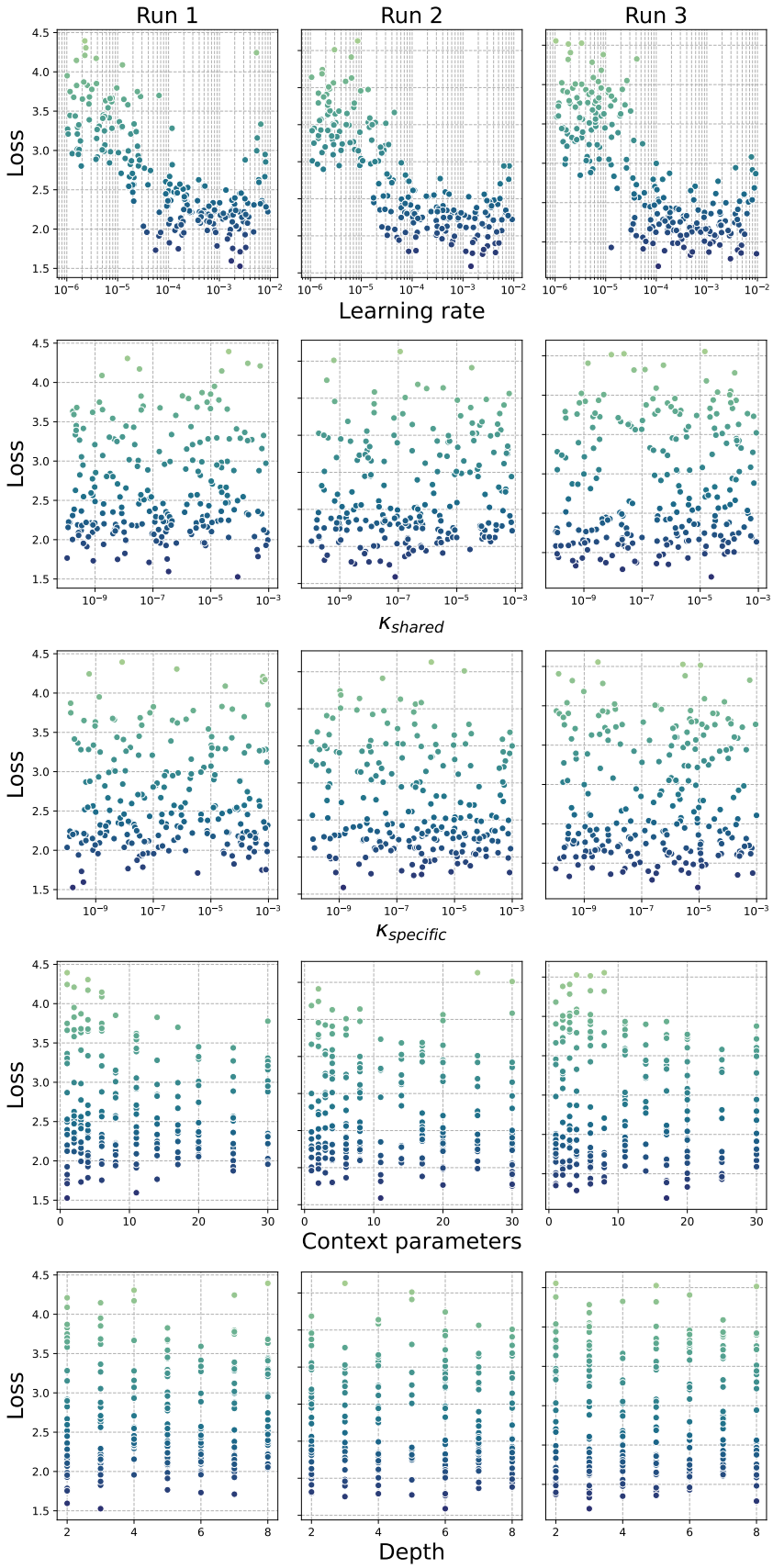


Figure C.2: Scatter plot of hyperparameters from individual runs of RS-MSR and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values.

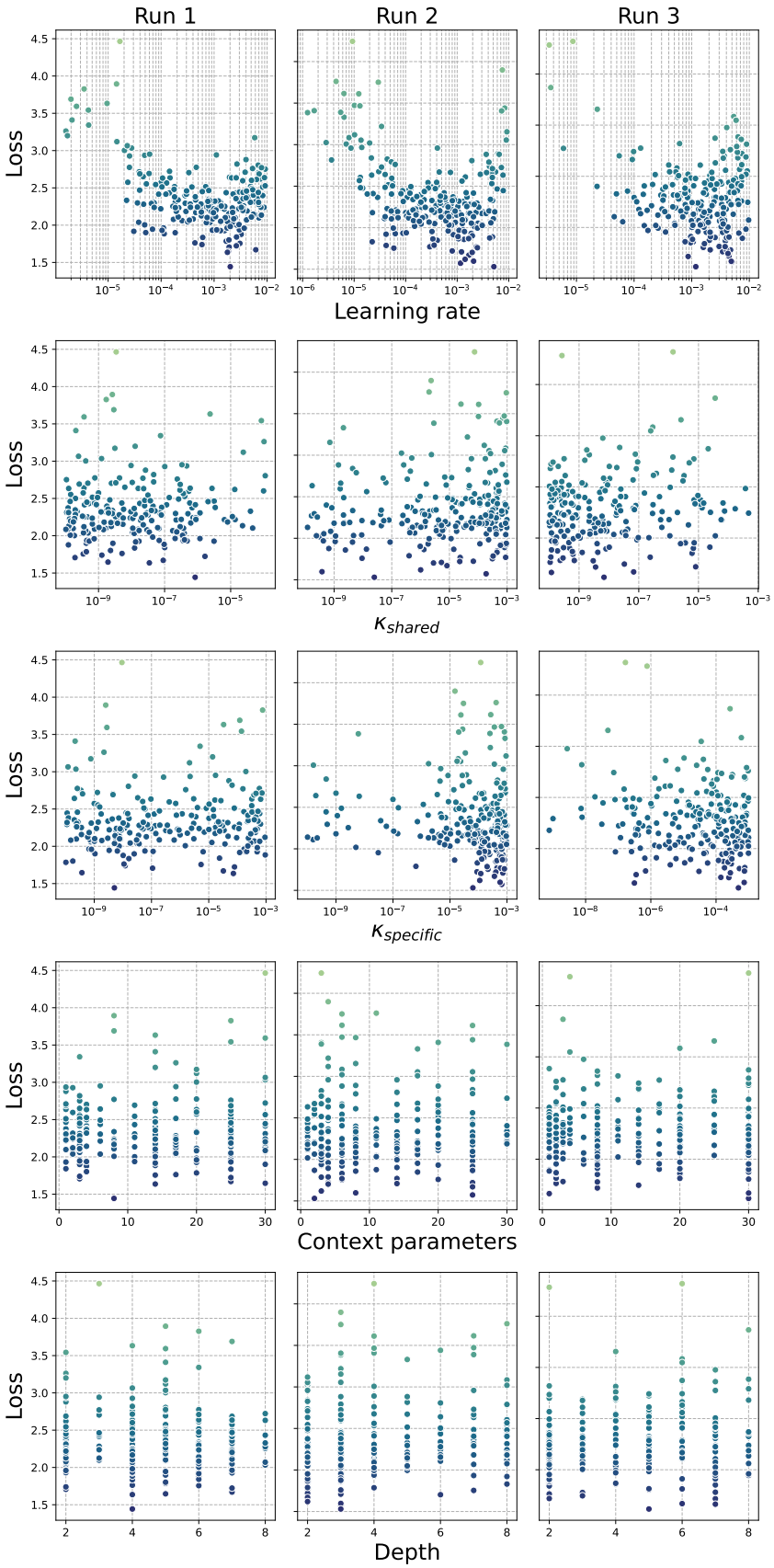


Figure C.3: Scatter plot of hyperparameters from individual runs of BO and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values. Notice how each run favours to sample hyperparameters from different regions of the configuration space, especially κ_{shared} and $\kappa_{specifc}$.

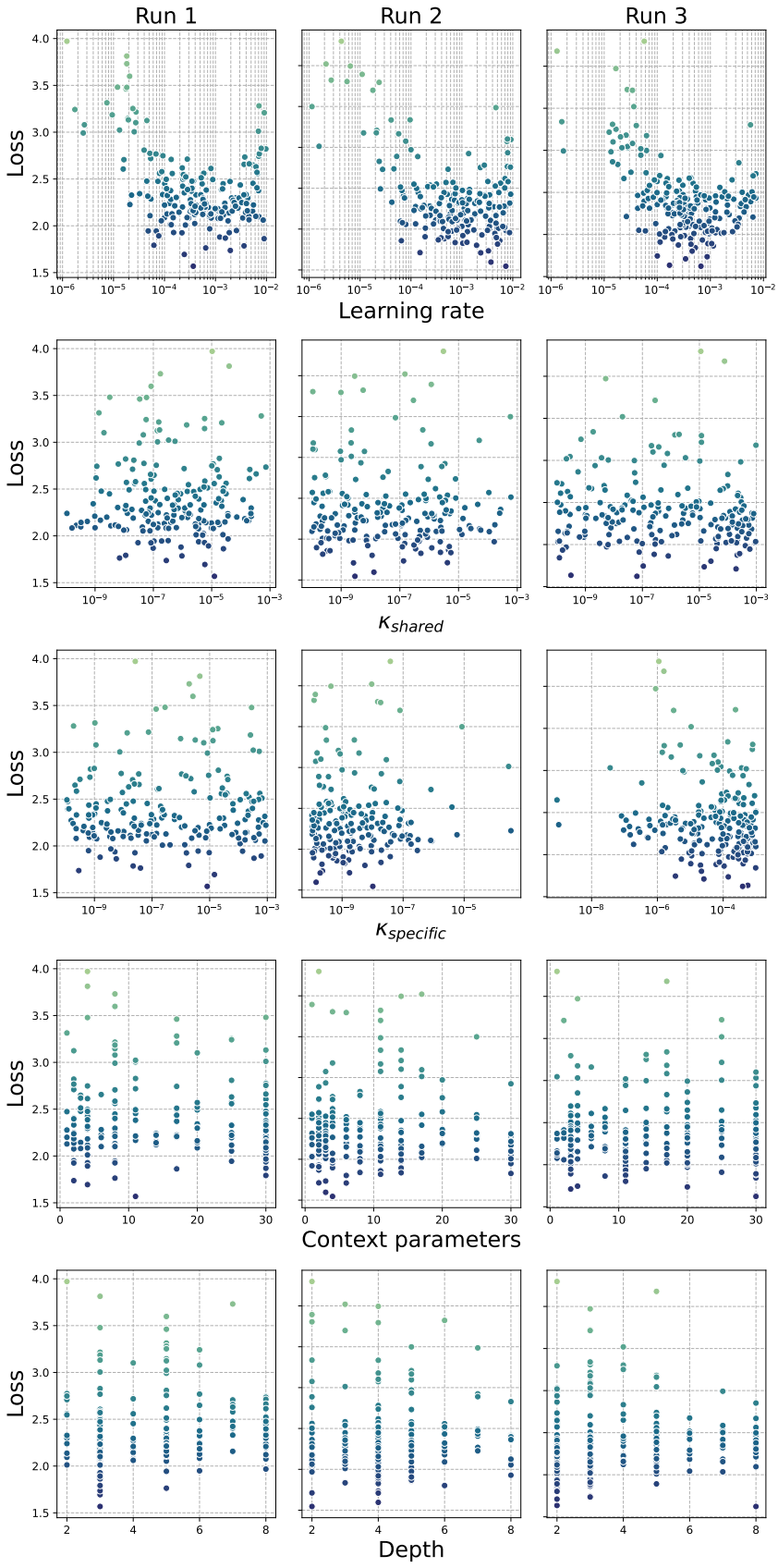


Figure C.4: Scatter plot of hyperparameters from individual runs of BO-MSR and corresponding loss. Trials with loss greater than 5 are left out for illustrative purposes. Only the learning rate seems to have a clear region of good and bad performing values. Notice how each run favours to sample hyperparameters from different regions of the configuration space, especially $\kappa_{specifc}$.

Knut Vågnes Eriksen

Hyperparameter Optimization for Neural Network-based Virtual Flow Metering

NTNU
Norwegian University of
Science and Technology