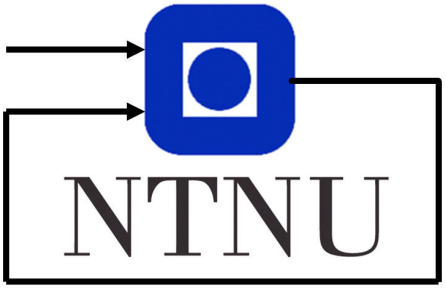


Department of Engineering Cybernetics

TTK4550 - Specialization Project

Exploring Hyperparameter Optimization for Neural Network-based Virtual Flow Metering

Author:
Knut Vågnes Eriksen

Supervisor:
Associate Professor Bjarne Andre Grimstad

Co-supervisor:
Professor Lars Struen Imsland

16th December 2021

Abstract

Hyperparameter Optimization (HPO) has recently been applied to advance state-of-the-art performances for many *Machine Learning* (ML) tasks. One particular HPO method that has gained widespread recognition is *Bayesian Optimization* (BO) due to its efficient acquisition of hyperparameters through the use of active modelling of expensive black-box functions. This modelling is typically done with the use of *Gaussian Processes* (GPs) that fits the black-box function as a joint multivariate Gaussian distribution.

This project explores the theoretical and practical world of HPO, with a particular focus on BO with GPs, to assist Solution Seeker advance the performance of their *Virtual Flow Meter* (VFM). Through a case study, the applicability of BO with GPs for a *Deep Neural Network* (DNN) within a computational budget is investigated, as well as the importance of the size of the hyperparameter configuration space. The results from the study show promising tendencies, where the BO on average performed as well as – or better than– a naive comparison method no matter the given computational budget or size of configuration space.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Source Code

Nomenclature

Abbreviations

Adam	Adaptive Moment Estimation
BO	Bayesian Optimization
CS	Configuration Space
DFFNN	Deep Feed-Forward Neural Network
DNN	Deep Neural Network
EI	Expected Improvement
GP	Gaussian Process
GP LCB	Gaussian Process Lower Confidence Bound
GP UCB	Gaussian Process Upper Confidence Bound
GPR	Gaussian Process Regression
GS	Grid Search
HPO	Hyperparameter Optimization
ML	Machine Learning
MLA	Machine Learning Algorithm
MSE	Mean Square Error
PI	Probability of Improvement
ReLU	Rectified Linear Unit activation function
RS	Random Search
SMBO	Sequential Model-Based Optimization
STD	Standard Deviation
TPE	Tree-structured Parzen Estimator
VFM	Virtual Flow Meter

Symbols

α	Acquisition function
$\boldsymbol{\lambda}$	A vector of hyperparamters

$\boldsymbol{\lambda}^*$	The optimal vector of hyperparameters
$\boldsymbol{K}$	Covariance matrix
Δ	percentage-wise change
η	Weight regularization paramter
$\Gamma(\cdot)$	Gamma function
$\kappa(\cdot)$	Covariance function
Λ	Overall hyperparameter configuration space
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 $\boldsymbol{\lambda}$
$\mathcal{H}$	History of previos observations
$\mathcal{N}(\mu, \sigma^2)$	The Gaussian distribution with mean μ and variance σ^2
μ	Mean
$\nabla_\theta J(\theta)$	Gradient of J with respect to θ
ω	Exploitation-Exploration trade-off parameter
Φ	Standard normal cumulative distribution function
ϕ	Standard normal density function
ψ	Learning rate
Σ	Covariance
τ	Current best solution
θ	Neural Network Weight
ζ	Activation function
D_{Test}	Test set
D_{Train}	Training set
D_{Valid}	Validation set
E	Error
$H_\nu(\cdot)$	Bessel function of order ν
$J(\cdot)$	Cost function
$m(\cdot)$	Mean function
$V(\cdot)$	Validation error
M	Surrogate Model

1 | Introduction

1.1 Motivation

The use and development of fossil fuels have since their discovery been a key part of the world's rapid industrialization and development. Oil and gas accounted for 55.9% of the global energy mix in 2020, whereas hydroelectricity and other renewable energy sources only accounted for 12.6% (BP, 2021). Although fossil fuels still are the primary energy source in the world, oil and gas production has become increasingly more complex throughout the years, drilling deeper and in more remote places than ever. This field has typically been driven by classical engineering with fundamentals in mathematics, physics, and chemistry. Over the recent years, this trend has somewhat changed due to the fourth industrial revolution. With data being sampled at a rate that we have never seen before, the use of artificial intelligence has gained an increasing interest in the industry.

Specifically, breakthroughs in *machine learning algorithms* (MLA), deep learning and the use of GPU's for computational power has made artificial intelligence a powerful and applicable tool. Solution Seeker is a Norwegian company that harvests the powers of this tool. They have developed the world's first artificial intelligence for production optimization in the oil and gas industry (Solution Seeker, 2021), with use cases extending to test optimization, well monitoring and flow rate predictions.

This exploratory work is done in collaboration with Solution Seeker and focuses on the latter use case; flow rate prediction. Traditionally, intricate multiphase flow meters have been used to measure the flow rate. Thorn et al. (2012) highlights some importance of three-phase flow metering; They describe multiphase flow metering as a key part in making small petroleum fields cost-effective, reducing pipeline costs and making the development of new fields robust to fluctuating oil and gas prices.

Designing and building instruments that are capable of measuring multiphase flow is however a difficult science, as Falcone (2009) summarizes; 'So far, many alternative metering systems have been developed, but none of them can be referred to as generally applicable or universally accurate.' Because multiphase flow metering is such a challenging problem, Solution Seeker aims to provide the industry with data-driven models that can predict the flow rate through measurements that are more robust and easily obtained.

The company have acquired vast knowledge in designing data-driven models, called Virtual Flow Meters, for multiphase flow measurement in oil and gas production (Grimstad et al., 2021). Designing the VFM is one part of the challenge with ML. Another is extracting the optimal performance from the design. This work targets the latter problem by the use of hyperparameter optimization.

1.2 Background

The problem of HPO can be dated back to at least the 1990s. Kohavi and John (1995) showed that different hyperparameters led to different results for different data sets. Recently, HPO has gained an increasing interest in scientific research with the advancements of deep learning. Bergstra, Bardenet et al. (2011) argues among other things that the difficulty of selecting good hyperparameters makes it difficult to reproduce scientific results, and that improved state-of-the-art performances sometimes are due only to the selection of better hyperparameters rather than to innovation in machine learning strategies. Feurer and Hutter (2019) highlights some of the many use cases HPO has, and this work is in itself an example of the recognition HPO is gaining in the industry.

Manual hyperparameter tuning by experts has been known to improve the performance of MLA's. This method is however inefficient, and hyperparameter expertise gained on one data set in one ML domain is often not transferable to other data sets and domains. Naive methods such as *Grid Search* (GS) and *Random Search* (RS) have shown good results (Bergstra and Bengio, 2012), but are still not optimal. They showed by theoretical proof and empirical evidence that for neural networks, RS found 'models that are as good or better [than GS] within a small fraction of the computation time'.

Since resources like computation time, computational power, and money are restricted, HPO is rendered a crucial step in optimizing the learning and performance of machine learning algorithms. One of the most popular and recognized methods for HPO is Bayesian optimization (Brochu et al., 2010; Shahriari et al., 2015) with Gaussian processes (Williams and Rasmussen, 2006). BO has shown great results recently in many domains. Snoek, Larochelle et al. (2012) obtained state-of-the-art performance on the object recognition set CIFAR-10. Bergstra, Yamins et al. (2013) later used BO to optimize convolutional neural networks to obtain new state-of-the-art performance on CIFAR-10, as well as on a face identification task (PubFig83) and a face-matching verification task (LFW). Mendoza et al. (2016) used BO to win two competition data sets against human experts in the *ChaLearn AutoML Challenge* (Guyon et al., 2015) using automatically tuned feed-forward neural networks. Solution Seeker have identified HPO and BO as an interesting field to further develop their VFM.

1.3 Objectives and Research Questions

To assist Solution Seeker in their search for optimal hyperparameters, the following objectives were defined:

- The primary objective of this work is to form a solid theoretical and practical foundation within the fields of hyperparameter optimization, Bayesian optimization and Gaussian processes for machine learning. The experiences and reflections made through this work will establish the basis for the author's upcoming master thesis, which will dive deeper into the world of HPO.
- The secondary objective is to perform a brief software review of available HPO packages compatible with Python. The goal of the review is to identify well-tailored software packages to be used during the master thesis, and for further work at Solution Seeker.

The following research questions were defined to reach the primary objective:

1. Can Bayesian optimization outperform naive hyperparameter optimization techniques on a deep neural network given a computational budget?
2. What significance does the size of the configuration space have for the result?

The scope of this work is restricted to a simple case study. Answers to the research questions, results, discussions, and conclusions drawn in this work are thus merely an indication of the general picture.

1.4 Outline

Chapter 1 introduces this work. In Section 1.1 the background and motivation for this work is given. In Section 1.2 a brief literature review of some important contributions to HPO is given. In Section 1.3 the objectives of this work are defined, and the research questions asked to reach the objectives are stated.

Chapter 2 gives the theoretical knowledge needed for this work. In Section 2.1 a brief introduction to machine learning and deep learning is given. The supervised learning problem and the term hyperparameter are defined. In Section 2.2 the problem of hyperparameter optimization is defined. An introduction to model-free and model-based hyperparameter optimization is given. Section 2.3 builds on the introduction to model-based HPO and introduces Bayesian optimization. Section 2.4 gives a brief explanation of the surrogate model used for the Bayesian optimization in this work. In particular, it introduces Gaussian processes, *Gaussian Process Regression* (GPR) for machine learning, the challenge of scalable Gaussian processes, and different covariance functions used to describe the GP Section 2.5 highlights some of the most popular acquisition functions used for Bayesian optimization, and a small discussion on which acquisition function to use.

Chapter 3 introduces some practical aspects of BO. In Section 3.1 some typical hyperparameters for deep neural networks are given and their effect explained, along with recommendations for choosing appropriate search spaces. In Section 3.2 a brief software review is conducted.

Chapter 4 explains the case study. In Section 4.1 the case is introduced, and the data set for the case is described. In Section 4.2 the experiment designed to answer the research questions is argued for and described. Section 4.3 explains the Python implementation of the case. Chapter 5 gives the results of the experiment in Section 5.1, and discusses them in Section 5.2 Chapter 6 concludes the work in the form of a summary, along with recommendations for future work.

2 | Theory

2.1 Intro to Machine Learning and Deep Learning

2.1.1 Machine Learning Basics

Humans tend to seek patterns to obtain an order, but if the scope becomes too vast we are unable to process the information in a suiting manner. This is where machine learning comes to play. Kevin P. (2012) defines ML as

> A set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision-making under uncertainty.

This work uses machine learning for two purposes. First machine learning is used to fit a model that can predict the total flow rate of water, oil, and gas in a pipe, then, ML is used to model an uncertain function to optimize the flow rate model.

Machine learning is usually divided into two classes: *Supervised-* and *Unsupervised-*learning. The former class treats problems where the object is to learn patterns from input to output, while the latter focuses on discovering interesting patterns solely from an input. This work focuses on the first type. The supervised learning problem can be defined as follows:

Definition 2.1.1 (Supervised Machine Learning)**:**
Observing a training set $\mathcal{D} = \{(\boldsymbol{x}_i, f_i), i = 1 : N\}$ *where* $f_i = f(\boldsymbol{x}_i)$*, and given a test set* $\boldsymbol{X}_*$ *of size* $N_* \times D$*, the goal is to predict the function outputs* $\boldsymbol{f}_* = f(\boldsymbol{X}_*)$*.*

Definition 1 mentions a training- and test-set. These two sets come from a common set, the *data set*. With supervised learning, the data set consists of multiple inputs with corresponding labelled outputs. The input-to-output pattern in the data set is what the MLA tries to learn. A common practice with ML is to at least split this data set into two parts, a *training set* used for training, and a *test set* used for testing. While training the model to recognize patterns in the training set, the learning algorithm aims to minimize the *training error*, which is the error observed between the model's prediction and the expected outcome in the training set. The model is then evaluated on the test set and the *test error*, which is the error observed between the model's predictions and the expected outcome in the test set, is calculated. The idea behind machine learning, and what makes it different from an optimization problem, is that by minimizing the training error the test error is also minimized.

The test error is also called the *generalization error*, which is a measure that tells us how accurate the model can predict outcome values for unseen data. In other words, the test error tells us something about how well the model would perform if it was deployed in the real world, where all future data is currently unseen. The model's ability to predict outcomes from unseen data, and not only memorize the previously seen data, is called *generalization*. The difference between the test error and the training error is called the *generalization gap*.

To justify that minimizing the training error also minimizes the test error, an assumption that is typically referred to as the *i.i.d assumption* has to be made. The assumption holds if the samples in the data set are independent of each other, and that they are identically distributed, i.e. drawn from the same probability distribution. Since the i.i.d assumption is made, one would expect the expected training error to be equal to the expected test error. This is not the case, however, since the MLA minimizes the training error, and we would rather thus expect the test error to be greater or equal to the training error. Based on this, Goodfellow et al., 2016 defines the factors that determine how well a MLA performs to be its ability to:

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

These two abilities are analogous to the common ML terms *underfitting* and *overfitting*. Underfitting happens when the model is not able to make the training error small enough. Overfitting happens when the gap between training and test error is too large. None of these properties are desired, and we wish to avoid both. This can be controlled by the model's *capacity*, which loosely is the model's ability to fit a wide variety of functions (Goodfellow et al., 2016). An illustration of the relationship between training error, test error, underfitting, overfitting and capacity can be seen in Fig. 2.1.1.

A simple solution to finding optimal capacity would be to fit the machine learning model with the training data, evaluate how it performs on the test data, and then tweak the model, re-fit and re-evaluate until the training error and generalization gap is satisfying. However, this would lead to what is commonly referred to as *data leakage*, which is when data from outside the training set is used to guide the MLA. In fact, by doing as proposed, the model would be optimized on the training and the test set, which in turn means that nothing can be said about the test error, as none of the data in the data set is left unseen.

A popular way of avoiding the problem of data leakage is to split the data set even further by introducing a *validation set*, as illustrated in Fig. 2.1.2. Then instead of tweaking the model based on the test error, it is tweaked based on the *validation error*, which is the observed error between the model's prediction and the expected output in the validation set. This way the test set is left unseen and a valid estimate of the test error can be calculated. This work utilizes this technique to optimize the model's capacity and is discussed in Section 2.2.2.

To be able to perform the minimization of the training error at all, a *learning algorithm* that can recognize the patterns which we are looking for needs to be defined: *A learning algorithm is a set of instructions that tries to extract patterns from data in order to apply what is learnt to new data.* There exist many learning algorithms that serve different purposes, and for a quick introduction to many popular algorithms being used, see Brownlee (2019). As stated at the start of this section, this work uses ML for two different purposes; The MLA used for the first purpose is a deep neural network that predicts flow rates, and for the second purpose it uses a Gaussian process regressor to model a function that is used to optimize the DNN.

To understand how a DNN can learn patterns from a data set, an introduction to deep learning and neural networks is given in Section 2.1.2. How this neural network's capacity is optimized is described in Section 2.2 - 2.5. For further information about machine learning, artificial neural networks and deep learning, please see Kevin P. (2012) and Goodfellow et al. (2016).

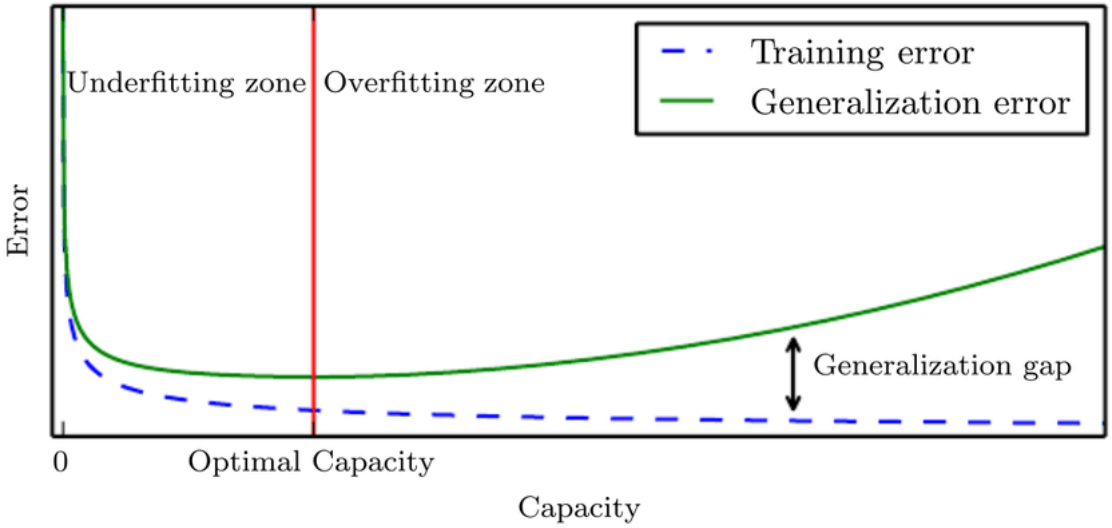


Figure 2.1.1: Capacity vs Overfitting & Underfitting. Illustration from Goodfellow et al. (2016)

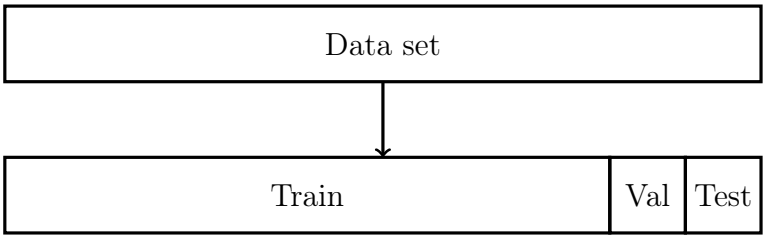


Figure 2.1.2: Data set separation.

2.1.2 Deep Learning and Neural Networks

An *artificial neural network* (often simply a neural network) is one MLA used for extracting patterns. It consists of multiple *neurons*, grouped in *layers*, that are connected. The layers are typically divided into three categories, *input*, *hidden* and *output*. The input and output layer refer to the first and last layer of the network respectively, while a hidden layer is any layer between the input and the output layer. Each neuron has its own *activation*, and each connection between a neuron n_i^{L-1} in layer $L-1$ and n_j^L in layer L has its own associated weight (θ_{ij}^L). A conceptual way to think about the activation of a neuron, is that if e.g. neuron n_i^{L-1} is activated, it is allowed to pass its weighted connection θ_{ij}^L forward to neuron n_j^L, and if it is not activated, no weights are passed forward. A neuron's activation is determined by its associated nonlinear *activation function*, $\zeta(\cdot)$, that determines the output of the neuron based on its input. This nonlinearity is what allows the neural network to recognize non-trivial patterns.

Furthermore, a neural network is given the term *feed-forward* if there are no feedback connections from one layer to another, and it is called *deep* if there are two or more hidden layers. An illustration of a deep feed-forward neural network (DFFNN) can be seen in Figure 2.1.3. The output, $\boldsymbol{y}$, of a DFFNN, is a composite mapping of the activation function in the different layers from the input $\boldsymbol{x}$. The output of the network in Fig. 2.1.3 can thus be written as:

$$\boldsymbol{y} = (\zeta^3 \circ \zeta^2 \circ \zeta^1)(\boldsymbol{x}),$$

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

Tweaking $\boldsymbol{\theta}$ in the different layers is what allows the neural network to recognize the patterns in the data set at hand. How these weights are tweaked is controlled by a forward-backwards-propagation scheme and the use of *gradient descent* methods. *Forward propagation* refers to giving the network an input that is propagated forwards through the layers of the network and finally produces an output. The output is then used to calculate the training error, a scalar cost $J(\boldsymbol{\theta})$, by the use of a *cost function* $J(\cdot)$. *Back-propagation* then allows the cost to propagate backwards through the network to compute the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. The weights are then updated through gradient descent as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \psi \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k), \tag{2.1}$$

where ψ is the step length, or *learning rate* in machine learning terms. For more on back-propagation, please see Goodfellow et al. (2016, Section 6.5) and for gradient descent, see Goodfellow et al. (2016, Section 4.3) or Ruder (2017).

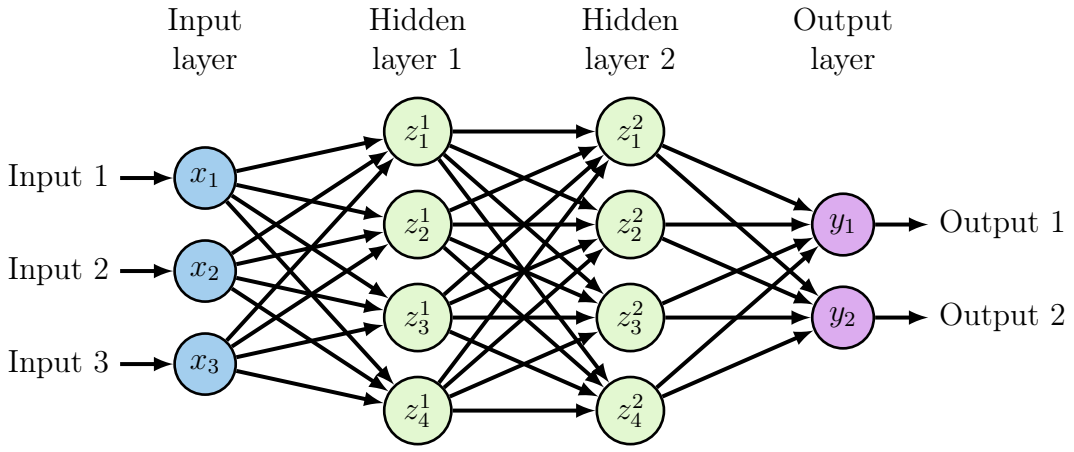


Figure 2.1.3: Deep Feed-Forward Neural Network with 3 inputs, 2 hidden layers and 2 outputs. The nodes represent neurons and the vertices represent weighted connections.

2.1.3 Hyperparameters

The number of hidden layers, number of neurons and the step length are examples of *hyperparameters*. A hyperparameter is a parameter (or a setting) that is used to control the learning process and behaviour of a MLA. In contrast to the weights mentioned in the previous section, these parameters are configured beforehand and are not learnt or updated throughout the process of training.

Compared to the parameters learnt by the machine learning algorithm that are typically real-valued between 0 and 1, hyperparameters have many domains. They can be 'real-valued (e.g. learning-rate), integer-valued (e.g. number of hidden layers), binary(e.g. whether to use early stopping or not), categorical (e.g. choice of optimizer)', or conditional (only use this hyperparameter if some condition is true) (Feurer and Hutter, 2019).

There exist many hyperparameters, and some typical hyperparameters for deep neural networks are given Section 3.1. To optimize the DNN's capacity we will try to select an optimal set of these hyperparameters as described in the following section.

2.2 Hyperparameter Optimization

Since the hyperparameters are what controls the behaviour of the MLA, it almost goes without saying that selecting good hyperparameters is crucial for obtaining an adequate model. Optimization of an *objective function* $f(\boldsymbol{x})$ over a compact set $\mathcal{D}$ is often formulated as:

$$\max_{\boldsymbol{x}\in\mathcal{D}} f(\boldsymbol{x})$$

With HPO we are rather concerned with minimizing a function. In particular, we are interested in minimizing the validation loss of a DNN as a function of the model's

hyperparameters. Feurer and Hutter (2019) provides a well-written formulation of the problem rendered here:

Definition 2.2.1 (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 $\Lambda = \Lambda_1 \times \Lambda_2 \times \dots \Lambda_N$. A vector of hyperparameters is denoted by $\boldsymbol{\lambda} \in \Lambda$, and $\mathcal{A}$ with its hyperparameters instantiated to $\boldsymbol{\lambda}$ is denoted by $\mathcal{A}_{\boldsymbol{\lambda}}$.

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

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

where $V(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})$ 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} with cost function $\mathcal{L}$.

The objective is thus a functional from hyperparameters to validation loss. This problem is considered as black-box optimization, since the exact relationship between the input (hyperparameters), and the output (validation error) cannot analytically be expressed. In addition, the response-surface of $V(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})$ cannot be assumed to be convex, and neither that its gradient provide any information. Furthermore, the only way of gathering information about the objective is by sampling output values, which may be noisy. Sampling output values requires that the neural network is trained, and its predictions validated. Algorithms for solving these kinds of black-box optimization problems are typically divided into two categories; *model-free* and *model-based*.

2.2.1 Model-Free Hyperparameter Optimization

Model-Free algorithms are algorithms that do not try to model the black-box function, but rather just try out different configurations, hoping that one of them results in a good enough solution. Grid search is the simplest version of these algorithms. It requires that a set for each hyperparameter in Λ is chosen. Grid search then goes through every possible combination of these selected sets, in order to try to optimize $\boldsymbol{\lambda}$. This requires $\prod_{n=1}^{N} |\Lambda_n|$ iterations, which easily can be parallelized since there is no connection between different iterations. However, parallel processes have to communicate with each other to avoid selecting the same hyperparameters as other processes have chosen before. A problem with grid search is that it suffers from the *curse of dimensionality*, as the number of iterations grows exponentially with the number of hyperparameters (Bellman, 2015).

Another drawback with grid search comes from the fact that some hyperparameters often are more important than others (Kohavi and John, 1995; Bergstra and Bengio, 2012), and with pre-configured sets of Λ grid search effectively searches

over less of the important hyperparameters than actual iterations. An illustration of this phenomenon is shown in Fig. 2.2.1. Random search is an attempt to solve this problem. It draws different sets of Λ from a probability distribution, rather than from a pre-selected set of values. Thus, it evaluates random sets of Λ instead of pre-configured combinations of $\Lambda_1 \dots \Lambda_N$. Bergstra and Bengio (2012) showed that random search performs better than grid search given the same computational budget. In addition, RS is easier to parallelize than GS because there is no need for synchronization between processes and each trial is just as independent as with GS.

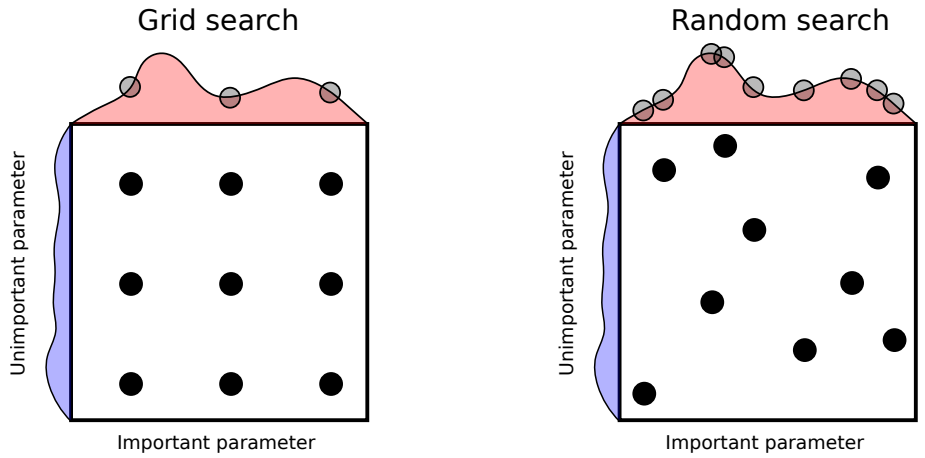


Figure 2.2.1: Illustration of grid and random search. The figure illustrates how random search is able to sample more of the important hyperparameters than grid search is. The figure was made based on Figure 1 in Bergstra and Bengio (2012).

2.2.2 Model-Based Hyperparameter Optimization

Instead of searching for the solution naively, these algorithms try to model the objective function in order to make informed choices for where to sample next. These methods are specifically designed for expensive objective functions, such as a neural network with many hyperparameters.

In particular, we consider Sequential Model-Based Optimization (SMBO). These algorithms use a *surrogate* to model the black-box function based on the inputs (previously tried sets of hyperparameters), and outputs (the validation loss corresponding to the sets of previously tried hyperparameters). Take e.g. the neural network that we to perform Bayesian optimization on later. Training the network is an expensive operation that requires a lot of resources. Thus, we want to limit the number of times we have to train the network as much as possible. Evaluating the surrogate on the other hand is cheap, and if necessary we evaluate this function many times to find an optimal solution. The key idea is that the vector $\boldsymbol{\lambda}^*$ that minimizes the surrogate M, also optimizes the machine learning algorithm $\mathcal{A}$ measured by the validation function $V(\cdot)$.

To do this the SMBO-algorithm sequentially solves for $\boldsymbol{\lambda}^*$ by the use of an *acquisition function*, α, trains the machine learning algorithm $\mathcal{A}_{\boldsymbol{\lambda}^*}$, compute the validation loss, updates the history of sampled values $\mathcal{H}$, fits the surrogate and then again solves for $\boldsymbol{\lambda}^*$. This process is given in the form of pseudo-code in Algorithm 1.

There are two key parts to the SMBO-algorithm. Deciding where to sample next (step 3 in Algorithm 1) is the first. This is done by choosing an acquisition function, sometimes referred to as a *utility function*, and maximizing this utility for the surrogate model. Different acquisition functions have been proposed, and some of the most popular ones for BO with GP are described in Section 2.5. The other key part is how the objective function is modelled (step 6 in Algorithm 1). There are numerous ways to do this, but this work focuses on using Bayes' rule, as described in the following section.

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

1: $\mathcal{H} \leftarrow \emptyset$
2: **for** $t \leftarrow 1$ **to** T **do**
3: $\quad \boldsymbol{\lambda}^* \leftarrow \text{argmin}_{\lambda}\, \alpha(\boldsymbol{\lambda}, M_{t-1})$
4: $\quad$ Train $\mathcal{A}_{\boldsymbol{\lambda}^*}$ and evaluate $V(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train}, D_{valid})$
5: $\quad \mathcal{H} \leftarrow \mathcal{H} \cup (\boldsymbol{\lambda}^*, V(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train}, D_{valid}))$
6: $\quad$ Fit a new model M_t to $\mathcal{H}$
7: **end for**
8: **return** $\mathcal{H}$

2.3 Bayesian Optimization

Bayesian Optimization is a SMBO algorithm that uses Bayes' rule to model the objective. Bayes' rule states that the *posterior* probability of a model M given observed data V is proportional to the *prior* probability of M multiplied by the *likelihood* of the observed data V given the model M:

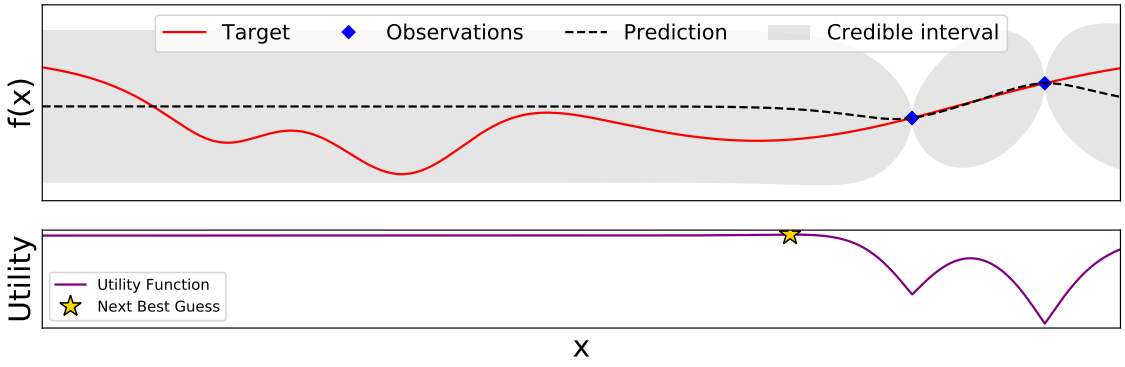
$$\begin{aligned} P(M|V) &\propto P(M) \times P(V|M) \\ \text{posterior} &\propto \text{prior} \times \text{likelihood} \end{aligned} \tag{2.2}$$

The *prior distribution* expresses the beliefs about the distribution of objective functions before any data is observed, i.e. which objective functions do we think are plausible? As mentioned before, the objective function is a black box, but we make some assumptions about it to guide the optimization. Combined with the likelihood, which modifies the beliefs about the prior according to the observed data, we obtain a *posterior distribution*. The posterior distribution is the actual distribution that is used to model the objective, after taking the observed data into account. Pseudo-code for BO is given in Algorithm 2. An illustration of the BO process is given in Fig. 2.3.1.

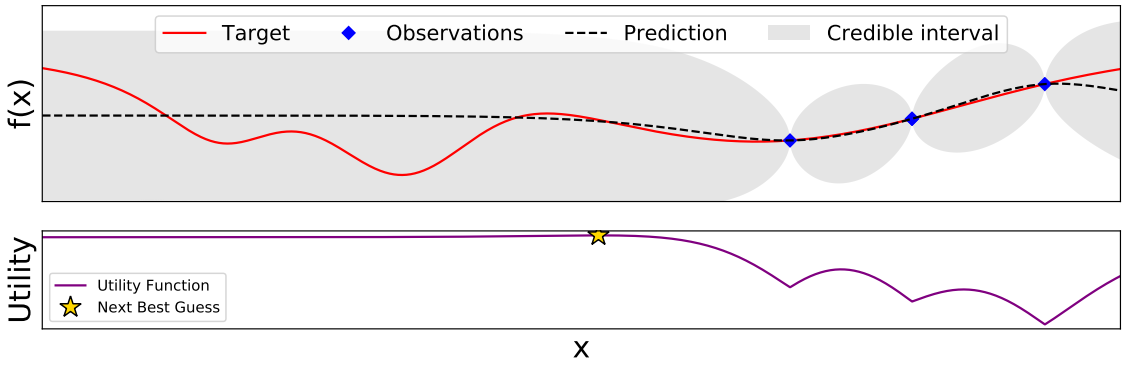
There exists different ways of obtaining the posterior distribution, or rather different priors you can apply to express the prior beliefs. Since data have to be sampled, and we often do not know much about the likelihood of the observed data, we often assume that the observations have a zero-mean Gaussian noise as described in Section 2.4.2. To be able to use the version of Bayes rule as described in Eq. (2.2), a conjugate prior has to be used. With a zero-mean Gaussian noise added to the likelihood, this conjugate prior corresponds to a Gaussian prior. This work focuses on expressing this prior with a Gaussian process, which is described in the following section. For more on Bayesian optimization, please see Brochu et al. (2010) and Shahriari et al. (2015).

Algorithm 2 Pseudo-code for Bayesian Optimization for Machine Learning Algorithms. Adapted from Bergstra, Bardenet et al. (2011)

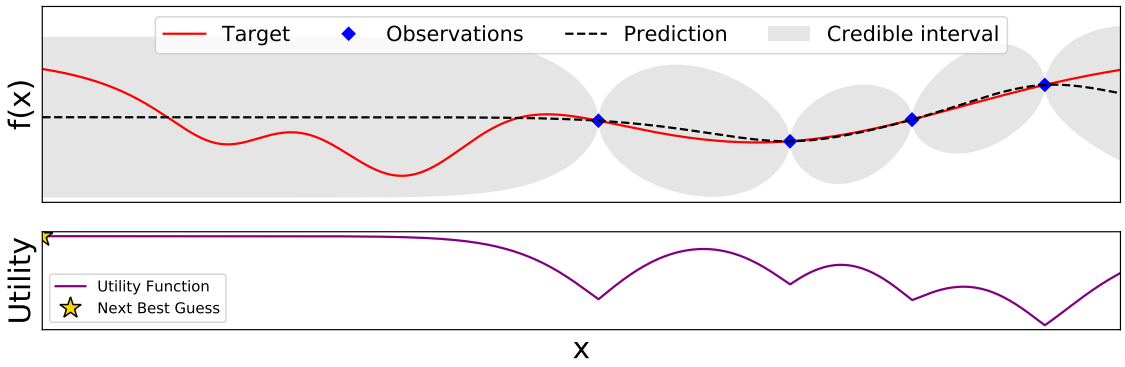
$\mathcal{H} \leftarrow \emptyset$
for $t \leftarrow 1$ **to** T **do**
    $\boldsymbol{\lambda}^* \leftarrow \text{argmin}_\lambda \, \alpha(\boldsymbol{\lambda}, P(f|\mathcal{H}_{t-1}))$
    Train $\mathcal{A}_{\boldsymbol{\lambda}^*}$ and evaluate $V(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train}, D_{valid})$
    $\mathcal{H} \leftarrow \mathcal{H} \cup (\boldsymbol{\lambda}^*, V(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train}, D_{valid}))$
    Update posterior distribution $P(f|\mathcal{H}) \propto P(f) \times P(\mathcal{H}|f)$
end for
return $\mathcal{H}$



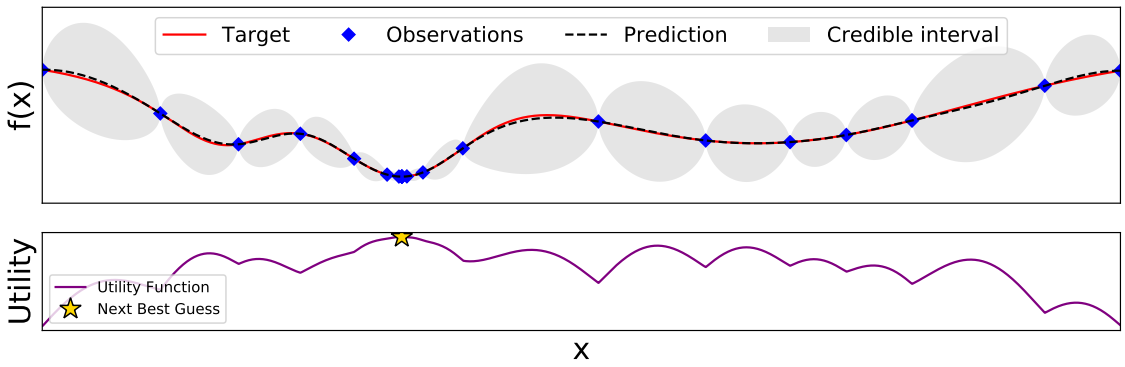
(a) After two random samples



(b) After one acquisition



(c) After two acquisitions



(d) After 18 acquisitions

Figure 2.3.1: Process of Bayesian optimization illustrated for a 1-d function. Credible interval shows 2 times STD. The star signifies where the algorithm has decided to sample next. Notice the dense observations near the minima in Fig. 2.3.1d, and that the acquisition function has decided that this is the convergence point. The figures were adapted from Nogueira (2014).

2.4 Surrogate Model

Gaussian Process Regression is a way of modelling functions using Bayesian statistics and regression analysis. The method combines Gaussian Processes with Bayesian inference in an attempt to describe the objective. In this section, a short introduction to GP and GPR is given, before the issue of scalable GP is discussed and some popular covariance functions are introduced. Section 2.4.1 and Section 2.4.2 is widely adapted from Williams and Rasmussen (2006).

2.4.1 Gaussian Processes

Williams and Rasmussen (2006) defines a GP as follows:

> A Gaussian process *is a collection of random variables, any finite number of which have a joint Gaussian distribution.*

In other words, a GP can be seen as a Gaussian distribution over functions, just as a Gaussian distribution is a distribution over a random variable. Furthermore, and similar to a Gaussian distribution that is completely specified by its mean and covariance, the GP ($f(\boldsymbol{x})$) is a non-parametric model that is entirely defined by its mean function; $m(\boldsymbol{x})$, and covariance function (kernel); $\kappa(\boldsymbol{x}, \boldsymbol{x'})$:

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), \kappa(\boldsymbol{x}, \boldsymbol{x}')) \tag{2.3a}$$
$$m(\boldsymbol{x}) = \mathbb{E}[f(\boldsymbol{x})] \tag{2.3b}$$
$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x}') - m(\boldsymbol{x}'))] \tag{2.3c}$$

2.4.2 Gaussian Process Regression

A Gaussian process regressor uses Bayes' rule with a GP prior to obtain a fit posterior distribution of the objective function. To see how this is done, the supervised learning problem (Definition 2.1.1) is reformulated to fit the specific problem at hand:

Definition 2.4.1 (Gaussian Process Regression)**:**
Given a history set $\mathcal{H} = \{(\boldsymbol{\lambda}_t, V_t), \quad t = 1 : T\}$ *of previous observations where* $V_t = V([\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}_t}, D_{train}, D_{valid}])$*, the goal is to predict the function outputs* V_{T+1}*.*

With traditional Bayesian inference, the goal would be to calculate a probability distribution of $\boldsymbol{\lambda}$ for a specific function V. Instead, GPR calculates the probability distribution of all possible functions that fit the observed data. By placing a Gaussian prior on V as illustrated in Fig. 2.4.1, the goal is to modify these priors to fit the objective function.

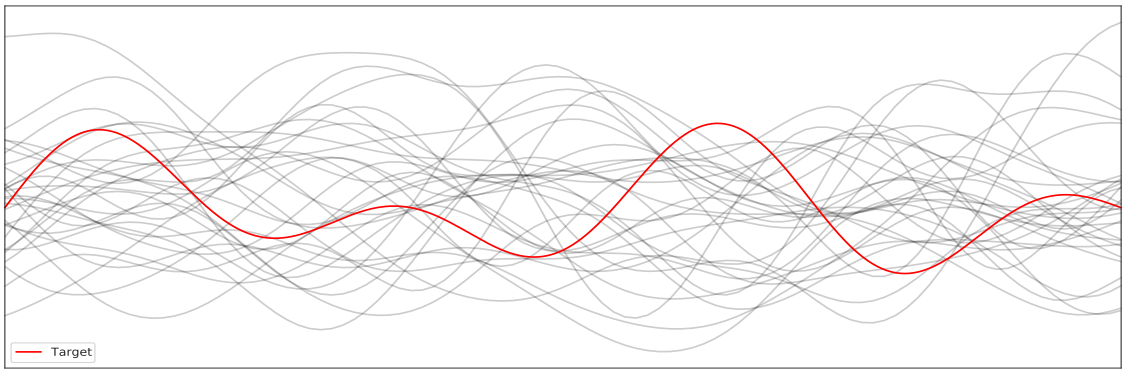


Figure 2.4.1: Samples from the prior distribution. Randomly selected priors are shown in light grey, with the target depicted in red. The goal with Gaussian process regression is to sample data points from the target, such that the posterior (i.e. the modified priors) eventually fit the target.

Since a GP prior is used, $V_{1:T}$ and V_{T+1} can by the properties of Eq. (2.3) be modelled as a joint Gaussian distribution. Defining $\boldsymbol{x}_t = [\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}_t}, D_{train}, D_{valid}]$ and $\boldsymbol{X}_t = [\boldsymbol{x}_t]$ the distribution reads:

$$\begin{bmatrix} V_{1:T} \\ V_{T+1} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\boldsymbol{X}) \\ m(\boldsymbol{X}_*) \end{bmatrix}, \begin{bmatrix} \boldsymbol{K} & \boldsymbol{K}_* \\ \boldsymbol{K}_*^\intercal & \boldsymbol{K}_{**} \end{bmatrix} \right), \tag{2.4}$$

where $\boldsymbol{K} = \kappa(\boldsymbol{X}_{1:T}, \boldsymbol{X}_{1:T})$ is $\mathbb{Z}^{T \times T}$, $\boldsymbol{K}_* = \kappa(\boldsymbol{X}_{1:T}, \boldsymbol{X}_{T+1})$ is $\mathbb{Z}^{T \times 1}$, and $\boldsymbol{K}_{**} = \kappa(\boldsymbol{X}_{T+1}, \boldsymbol{X}_{T+1})$ is $\mathbb{Z}^{1 \times 1}$.

Obtaining the function output V_{T+1} can be achieved by conditioning (2.4) on the observations $V_{1:T}$, which results in the posterior distribution:

$$p(V_{T+1} | \boldsymbol{X}_{T+1}, \boldsymbol{X}_{1:T}, V_{1:T}) \sim \mathcal{N}(\boldsymbol{\mu}_{T+1}, \boldsymbol{\Sigma}_{T+1}) \tag{2.5a}$$

$$\boldsymbol{\mu}_{T+1} = m(\boldsymbol{X}_{T+1}) + \boldsymbol{K}_*^\intercal \boldsymbol{K}^{-1}(V_{1:T} - m(\boldsymbol{X}_{1:T})) \tag{2.5b}$$

$$\boldsymbol{\Sigma}_{T+1} = \boldsymbol{K}_{**} - \boldsymbol{K}_*^\intercal \boldsymbol{K}^{-1} \boldsymbol{K}_* \tag{2.5c}$$

By evaluating the mean, $\boldsymbol{\mu}_{T+1}$, and the covariance, $\boldsymbol{\Sigma}_{T+1}$, we can predict function values for V_{T+1}, as was the goal in Definition 2.4.1. The solution provided in (2.5) assumes data without noise, which is an assumption that in most cases is not applicable, as the true function values are often not known and data has to be sampled. Adding independent identically distributed Gaussian noise ϵ with variance σ_y^2 yields

$$\boldsymbol{y} = V(\boldsymbol{x}) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_y^2)$$

A popular way of simplifying the posterior calculations, without affecting the correctness of the model, is by setting the mean function (2.3b) of the prior to 0 by normalizing the data and centring it around 0. By doing this, we are left with the following prior:

$$\begin{bmatrix} \boldsymbol{y} \\ V_{T+1} \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} \boldsymbol{K_y} & \boldsymbol{K_*} \\ \boldsymbol{K_*^\intercal} & \boldsymbol{K_{**}} \end{bmatrix}\right) \tag{2.6}$$

where $\boldsymbol{K_y} = \boldsymbol{K} + \sigma_y^2 \boldsymbol{I}$

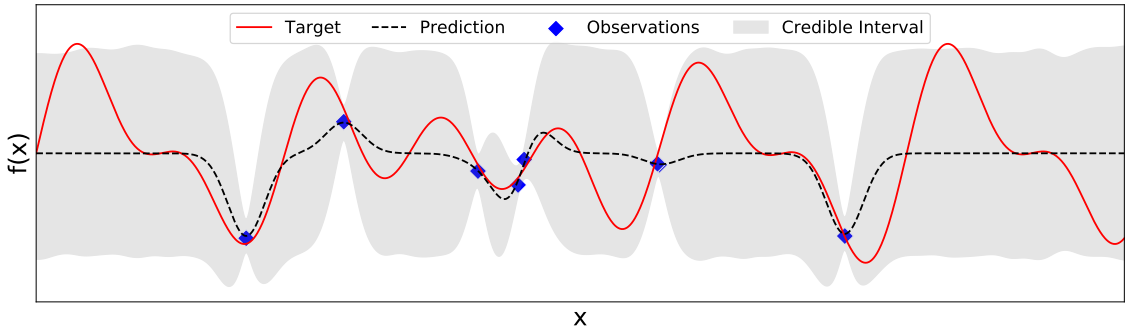
Again, conditioning Eq. (2.6) on the observations $\boldsymbol{y}$ yields:

$$p(V_{T+1}|\boldsymbol{X}_{T+1}, \boldsymbol{X}_{1:T}, V_{1:T}) \sim \mathcal{N}(\boldsymbol{\mu}_{T+1}, \boldsymbol{\Sigma}_{T+1}) \tag{2.7a}$$

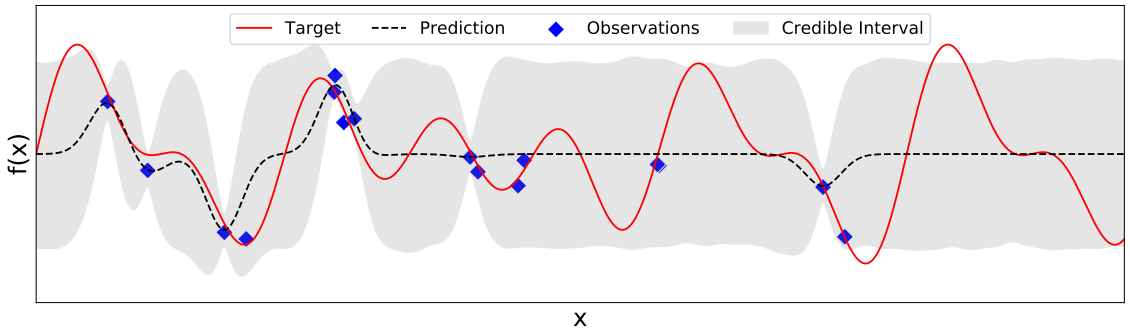
$$\boldsymbol{\mu}_{T+1} = \boldsymbol{K_*}^\intercal \boldsymbol{K_y}^{-1} \boldsymbol{y} \tag{2.7b}$$

$$\boldsymbol{\Sigma}_{T+1} = \boldsymbol{K_{**}} - \boldsymbol{K_*^\intercal} \boldsymbol{K_y}^{-1} \boldsymbol{K_*} \tag{2.7c}$$

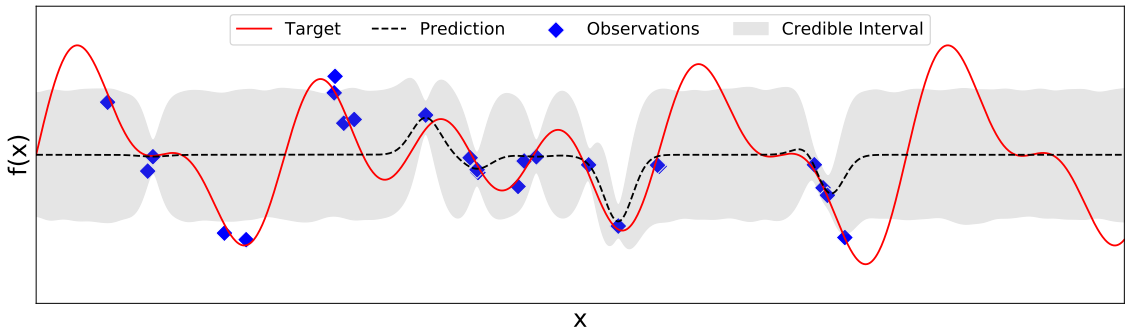
An illustration of the progress of GPR trying to fit a target function can be seen in Fig. 2.4.2. Notice the similarity between Fig. 2.3.1 and 2.4.2. The surrogate model in Fig. 2.3.1 was indeed a GPR, but while the goal with the optimization was to find the global minimum of the objective, the GPR tries to fit the entire function.



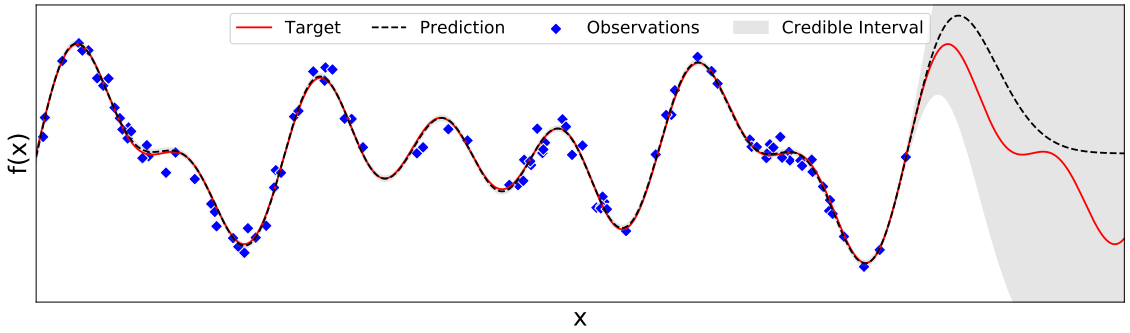
(a) Eight training points



(b) 16 training points



(c) 24 training points



(d) Showing 100 of 1000 training points

Figure 2.4.2: Illustration of the Gaussian Process Regression process. Credible interval shows 2 times STD. Fig. 2.4.2a shows the posterior distribution after fitting the prior to eight training points, resulting in an almost unison credible interval except just around the observations. The final result with almost a non-existing credible interval after fitting the prior to 1000 training points can be seen in Fig. 2.4.2d. Notice in Fig. 2.4.2d how poor the prediction is and how large the credible interval is. This is an illustration of the scalability issue with GPs. The figures were adapted from Orduz (2019).

2.4.3 Covariance Functions

As you can see from Eq. (2.6), the choice of covariance function κ plays an utmost important part. In fact, with a 0 mean, the kernel is what expresses our prior beliefs about the function which we wish to estimate. The kernel is constructed with the assumption that two points that are closer to each other in input space, have more similar function values than points that are further apart. Furthermore, it is required that the kernel is positive semidefinite. An explanation of the validity of covariance functions can be found in Williams and Rasmussen (2006).

The probably most used kernel is the *squared exponential kernel*:

$$\kappa_{se} = \exp\left(\frac{|\boldsymbol{x} - \boldsymbol{x'}|^2}{2l^2}\right), \tag{2.8}$$

where l defines the characteristic length-scale. This function is infinitely differentiable, and thus very smooth. In fact, it is so smooth that some (e.g Williams and Rasmussen (2006) and Snoek, Larochelle et al. (2012)) have recommended not using this function, but instead use the *Matérn* class of covariance functions:

$$\kappa_M = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{r\sqrt{2\nu}}{l}\right)^{\nu} H_\nu\left(\frac{r\sqrt{2\nu}}{l}\right),$$

where $r = |\boldsymbol{x} - \boldsymbol{x'}|$, ν and l are positive parameters, $H_\nu(\cdot)$ is the Bessel function of order ν and $\Gamma(\cdot)$ is the Gamma function. Varying ν there are infinite possible versions of the Matérn covariance function, but half-integer values of ν are the most popular ones, as the kernel is then a product of a polynomial and an exponential. It is argued in Williams and Rasmussen (2006) that $1/2 < \nu \leq 7/2$ are smart choices for ν, as for smaller values the process becomes very rough, and for larger values very hard to distinguish between values of ν. Williams and Rasmussen further suggest that $\nu = 3/2$ and $\nu = 5/2$ are possibly the most interesting cases for machine learning:

$$\kappa_{M3/2} = \left(1 + \frac{r\sqrt{3}}{l}\right)\exp\left(-\frac{r\sqrt{3}}{l}\right) \tag{2.9a}$$

$$\kappa_{M5/2} = \left(1 + \frac{r\sqrt{5}}{l} + \frac{5r^2}{3l^2}\right)\exp\left(-\frac{r\sqrt{5}}{l}\right) \tag{2.9b}$$

More kernels are discussed in Williams and Rasmussen (2006), but as suggested there they are not that appropriate for GPR as the kernels described above.

2.4.4 Challenges with Gaussian Processes

There are two major challenges with GPs: 1) Scalability and 2) Nonstationarity.

Scalability. Gaussian processes' scalability is limited by two factors. The first factor is that the hyperparameter configuration space can become too large, or the neural network too expensive to train. Then the GPR is not able to obtain enough observations within a reasonable amount of time. The posterior distribution is therefore not able to model the objective well enough for the acquisition function to give a valuable result. Thus, restricting the configuration space is essential for an efficient use of BO with GPs.

The second factor is the matrix inversion of $\boldsymbol{K_y}$ in Eq. (2.7b) and (2.7c). At iteration T, the inversion requires $\mathcal{O}(T^3)$ time. Thus, the time it takes to fit the model grows at a cubic scale every time a new point is sampled from the objective. This limits the use of GP's if obtaining many samples are inexpensive.

Attempts to overcome the challenge of scalability include the use of, but are not limited to, Tree-structured Parzen Estimators (TPEs) (Bergstra, Bardenet et al., 2011; Hutter et al., 2011), the use of deep neural networks (Snoek, Rippel et al., 2015) and Hyperband (Li et al., 2017; Falkner et al., 2018). The scope of these contributions reach outside the objectives of this work, and we leave the investigation of these methods for future work.

Nonstationarity. The kernels described in Section 2.4.3 assume that the underlying process is stationary, i.e. the covariance between two outputs is invariant to translation in input space. Hyperparameters have different domains with different length scales, and in addition, the configuration space that is searched over does not always follow a uniform distribution (Section 3.1. *Input warping* is a technique that is designed to overcome this issue, by replacing each input dimension with a Beta distribution. Even more special considerations have to be taken into account when using conditional configuration spaces. The *Arc* kernel uses a Euclidean distance space to accommodate the conditional configuration space for GP's. However, TPE's supports the varying length scales of different configuration spaces natively, which makes it even more interesting for future works. For more on this issue, please see Shahriari et al. (2015, Chapter V) and Feurer and Hutter (2019, Section 1.3.2.3).

2.5 Acquisition Functions

As mentioned in Section 2.2.2 there are two key aspects to SMBO; The surrogate model as described in Section 2.4 and the acquisition function to decide where to sample next. An important aspect of this acquisition function is the *exploitation-exploration trade-off* that we are faced with when given a decision-making problem. Do we want to exploit and continue searching for hyperparameters that we expect

will render a possibly slightly better performing neural network, or do we wish to explore and test hyperparameters that are unlike the ones we have tried before, in order to search more globally? In other words, do we wish to acquire points where the surrogate mean is low, or where the surrogate variance is high? An illustration of this trade-off can be seen in Fig. 2.5.2.

Generally, there are three acquisition functions used with BO and GPs: 1) Probability of Improvement, 2) Expected Improvement, and 3) GP Lower Confidence Bound. These three methods are introduced below. For more information, please see Jones (2001) and Brochu et al. (2010).

The *Probability of Improvement* (PI) maximizes the probability of improving from the current best solution $\tau = V(\boldsymbol{x}^-)$, where $\boldsymbol{x}^- = \text{argmin}_{\boldsymbol{x}\in\mathcal{H}} V(\boldsymbol{x})$:

$$PI(\boldsymbol{x}) = P(V(\boldsymbol{x}) \geq V(\boldsymbol{x}^-))$$

Since a GPR is used as the surrogate, and the posterior therefore is Gaussian, the probability of improvement is modelled as:

$$\alpha_{PI} = \Phi\left(\frac{\boldsymbol{\mu}_* - \tau}{\sqrt{\boldsymbol{\Sigma}_*}}\right),$$

where Φ is the standard normal cumulative distribution function.

This method is pure exploitation and selects points of high certainty that might give an infinitesimally better result over points that has a large variance but might give a much better result. A popular way of allowing this method to also explore is to add a trade-off parameter $\omega \geq 0$ to maximize the probability of improving at least ω:

$$\alpha_{PI} = \Phi\left(\frac{\boldsymbol{\mu}_* - \tau - \omega}{\sqrt{\boldsymbol{\Sigma}_*}}\right) \tag{2.10}$$

This allows for some control of the trade-off, but the method still favours exploiting, as it still selects the point where the probability of improving at least ω is highest. To overcome this, the method of expected improvement was proposed.

The *Expected Improvement* (EI) is somewhat similar to PI, except instead of trying to maximize the probability of improvement, the idea is to maximize how much improvement we can expect to achieve. The improvement is set to zero if there is no expected improvement, and it is negative when the prediction is lower than the best solution found so far. Sampling of the next point is done as follows:

$$\boldsymbol{x} = \underset{x}{\text{argmin}}\, \mathbb{E}(\min\{0, \boldsymbol{\mu}_* - \tau\}|\mathcal{H})$$

Again, since the posterior follows a Gaussian distribution the expected improvement is modelled as:

$$\alpha_{EI} = (\mu_* - \tau)\Phi\left(\frac{\boldsymbol{\mu}_* - \tau}{\sqrt{\boldsymbol{\Sigma}_*}}\right) + \sqrt{\boldsymbol{\Sigma}_*}\phi\left(\frac{\boldsymbol{\mu}_* - \tau}{\sqrt{\boldsymbol{\Sigma}_*}}\right),$$

where Φ is the standard normal cumulative distribution function, and ϕ is the standard normal density function.

As with PI, a trade-off parameter $\omega \geq 0$ is added to control the exploitation-exploration trade-off:

$$\alpha_{EI} = (\mu_* - \tau - \omega)\Phi\left(\frac{\boldsymbol{\mu}_* - \tau - \omega}{\sqrt{\boldsymbol{\Sigma}_*}}\right) + \sqrt{\boldsymbol{\Sigma}_*}\phi\left(\frac{\boldsymbol{\mu}_* - \tau - \omega}{\sqrt{\boldsymbol{\Sigma}_*}}\right) \tag{2.11}$$

The *GP Upper Confidence Bound* (GP UCB) proposed by Srinivas et al. (2009) is not improvement based as PI and EI. They proposed to treat the BO problem as a multi-armed bandit. The GP UCB aims to minimize the instantaneous regret function that measures the difference between the value of the point we choose to sample from, and the optimal solution:

$$r(\boldsymbol{x}) = V(\boldsymbol{x}^*) - V(\boldsymbol{x})$$

To do this, they defined the GP UCB as

$$\text{GP-UCB} = \boldsymbol{\mu}_* + \beta\boldsymbol{\Sigma}_*,$$

where β is a tunable hyperparameter. Instead of maximizing, we are interested in minimizing. Thus, we define the *Gaussian Process Lower Confidence Bound* (GP LCB) as:

$$\alpha_{LCB} = \boldsymbol{\mu}_* - \beta\boldsymbol{\Sigma}_* \tag{2.12}$$

Techniques for choosing β to achieve optimal regret are described in Srinivas et al. (2009). An illustration of the UCB and the LCB is shown in Fig. 2.5.1.

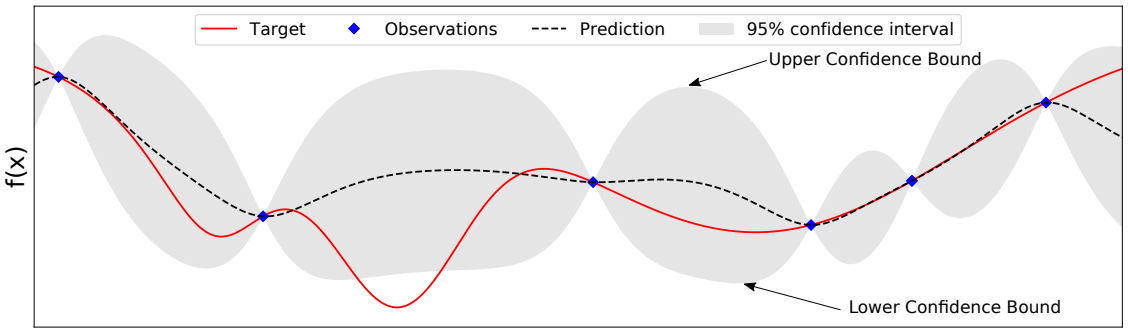
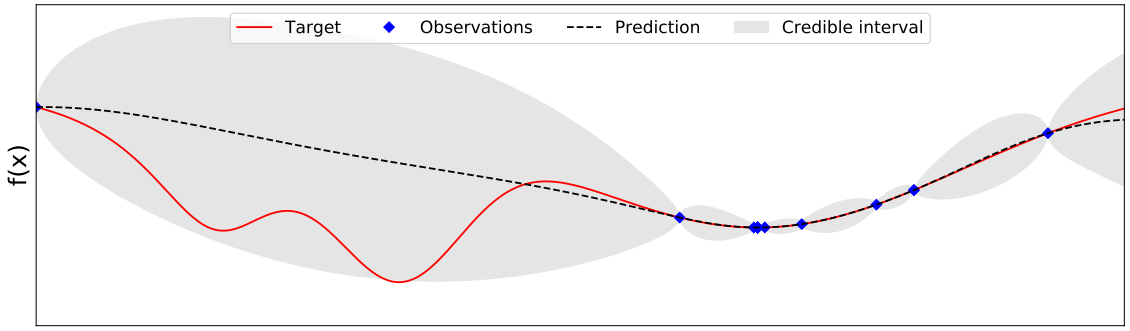
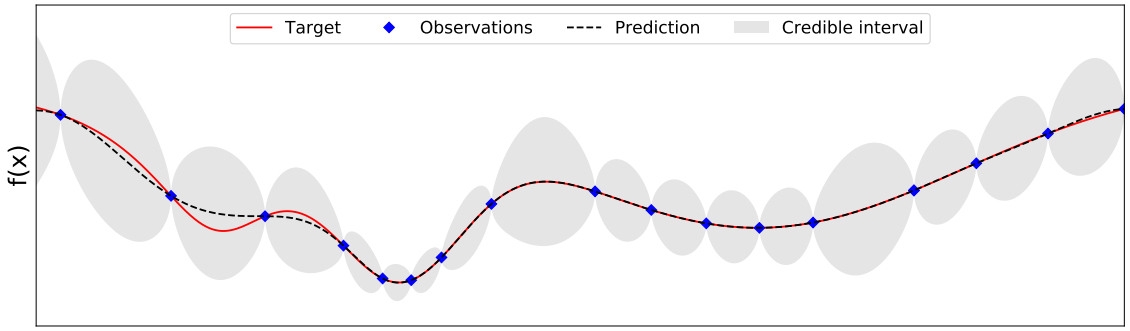


Figure 2.5.1: Illustration of confidence bounds. Notice the use of a confidence interval opposite to the credible interval used in other figures. Figure adapted from Nogueira (2014)



(a) Exploitation



(b) Exploration

Figure 2.5.2: Illustration of Exploitation vs Exploration. The figures were made using the GP-LCB acquisition function, but the concept applies to all acquisition functions mentioned in Section 2.5. In Fig. 2.5.2a it is illustrated that with too much exploitation, the Bayesian optimization converges towards a local minimum. In Fig. 2.5.2b on the other hand, we explore the areas with high variance too much such that the optimization method does not converge at all even though the algorithm has data from many samples close to the global minimum. The goal is to find a balance between these two extremes. Figures adapted from Nogueira (2014)

2.5.1 What to choose?

None of the proposed acquisition functions consistently provide better performance than the other. Nevertheless, the EI is generally preferred to PI as it allows for better control of the exploitation-exploration trade-off. When it comes to EI vs GP LCB nothing general can be said.

One interesting topic with acquisition functions is the control of the exploitation-exploration trade-off. Intuitively, one might think that adapting a method where exploration is preferred in the early stages and exploitation is preferred closer to the end of the optimization would result in better performance. However, this does not work well empirically (Brochu et al., 2010).

Another interesting topic is combining the use of different acquisition functions. E.g. Hoffman et al. (2011) showed that using a hedging strategy for a portfolio of acquisition functions, their method 'often performed substantially better – and almost never worse – than the best performing individual acquisition function'. We leave the investigation of portfolios of acquisition functions for further work.

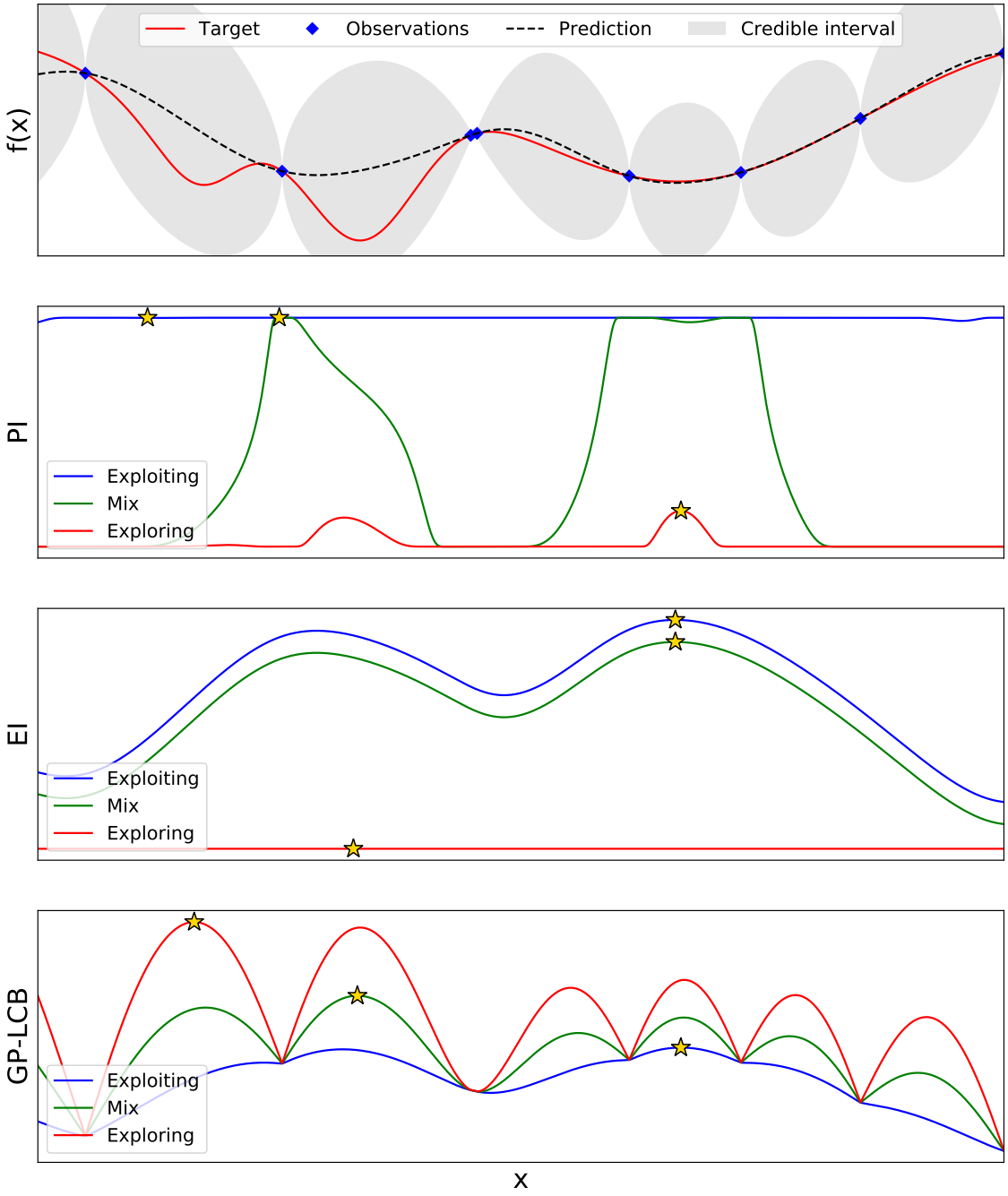


Figure 2.5.3: Comparison of PI, EI, and GP-LCB with different settings for controlling the exploitation-exploration trade-off. The credible interval shows 2 times STD. The next best guess is shown with a yellow star.

3 | Practical Bayesian Optimization

In this chapter, some typical hyperparameters for deep feed-forward neural networks are introduced, and some considerations that should be taken into account when selecting which hyperparameters to optimize are highlighted. Then, a brief review of available HPO software that supports the use of GP in Python is given.

3.1 Typical Hyperparameters

As mentioned earlier, there exist many hyperparameters depending on which learning algorithm you use. For DFFNNs, we define three categories of hyperparameters: 1) *Model Architecture and Complexity*, 2) *Training and Optimization* and 3) *Regularization*.

3.1.1 Model Architecture and Complexity

These hyperparameters affect the overall structure and complexity of the model. The *number of layers*, times the *layer width* (number of neurons in each layer), gives the total number of neurons in the network, and hence the complexity. More complex networks increase the representational capacity of the network and might result in better performance, but will again e.g. increase training time and memory usage. These two hyperparameters can also be combined to create different architectures for the network. Some popular styles are shown in Fig. 3.1.2.

A final model design hyperparameter is the activation function that determines how the neurons are activated. Generally speaking, we want differentiable non-linear activation functions to get the most out of the back-propagation algorithm. There exist many versions of these, and Nwankpa et al. (2018) provides a great summary for the interested reader. The *Rectified Linear Unit* (ReLU) function is usually a smart choice.

3.1.2 Model Training and Optimizer

First, the optimization referred to here is not to be mixed with the Bayesian optimization referred to earlier in this work. The model *optimizer* refers to the algorithm that is used to update the weights of the network while training, or in other words, different gradient descent methods. Ruder (2017, Section 4.10) recommends using an adaptive learning rate optimizer, as they are generally well-suited. Adaptive

Moment Estimation (Adam) (Kingma and Ba, 2017) is usually a clever choice.

The cost function which we use to measure the training and validation error also has to be selected. Regression problems, such as the one described in Section 4.1, generally use $\mathcal{L}_2$-loss functions on the form $||y' - y||_2^2$ (such as *Mean Square Error* (MSE)) as this corresponds to the maximum likelihood estimation for a linear regression model (Goodfellow et al., 2016, Section 5.5). If there are outliers in the data set, an $\mathcal{L}_1$-loss functions, $||y' - y||_1$, could perform better as outlier errors are not squared and given greater significance than necessary.

When updating the weights using the selected gradient descent method, we also have to define the learning rate. Too low a learning rate might result in slow convergence and overfitting, while too high might lead to divergent behaviours and underfitting. A simple illustration can be seen in Fig. 3.1.1. The learning rate is arguably the most important hyperparameter, as e.g. illustrated by Goodfellow et al. (2016, p.429): 'If you have time to tune only one hyperparameter, tune the learning rate.'

Two hyperparameters that greatly affect the time it takes to train a model is the number of *epochs*, and the *mini-batch size*. Epochs refer to the number of times each training point is passed through the network, while the mini-batch size refers to the number of training points used to make one step in the gradient descent. As a simple example, consider a training set of size $T = 1000$, epochs $n = 10$, and batch size $b = 50$. Training for 1 epoch then requires $T/b = 20$ gradient descent steps, and the whole training process requires $(T/b) * n = 200$ steps. Using a mini-batch size of n lowers the uncertainty in the gradient by a factor of $\mathcal{O}(\sqrt{n})$ but requires $\mathcal{O}(n)$ memory. In other words, larger batches might lead to a better step in the gradient descent, but in return requires more memory.

A final common technique used to speed up the learning process is called *batch normalization*, which refers to standardizing the output of a layer to a Gaussian distribution with zero mean and unit variance in each batch. Using batch normalization stabilizes the training and can significantly speed up the training process, as well as introducing some noise that can act as regularization (Goodfellow et al., 2016, Section 8.7.1).

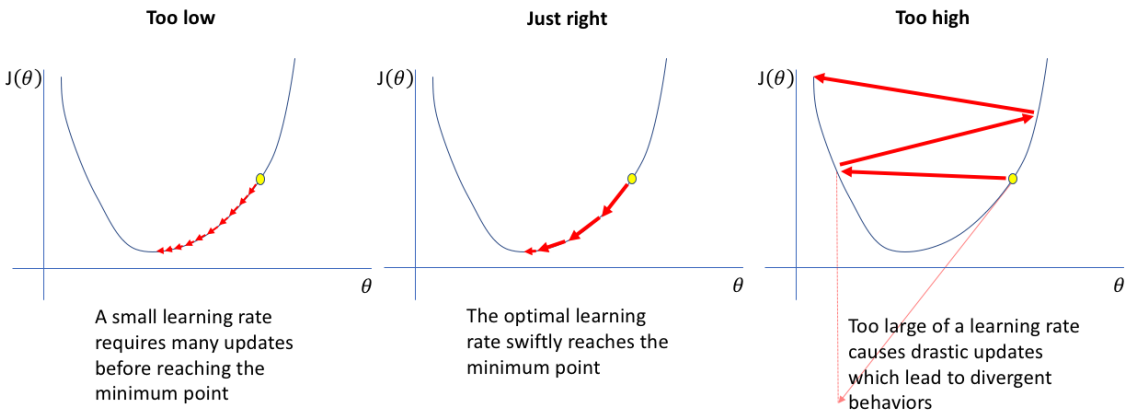


Figure 3.1.1: Learning rate. Picture from Jordan (2018)

3.1.3 Regularization

The final category of hyperparameters is hyperparameters that control the model's ability to reduce the possibility of overfitting. *Weight regularization* refers to adding a term to the cost function to penalize large model parameters. Drawing the larger weights closer to the origin results in a more well-informed gradient where no single parameter influences the step direction superiorly. $\mathcal{L}_2$ regularization is again generally preferred, but $\mathcal{L}_1$ has its uses as well. With a general cost function $J(\cdot)$, adding $\mathcal{L}_2$ regularization with regularization parameter η yields:

$$\hat{J}(\theta) = J(\cdot) + \eta||\theta||_2^2 \tag{3.1}$$

Another regularization method is *Dropout* (Srivastava et al., 2014) that uses samples from a Bernoulli distribution with a probability p of deactivating a neuron. The probability is referred to as the *dropout rate*. Dropout is an effective technique that prevents co-adaptation, which is when different hidden units have highly correlated behaviour. They propose a dropout rate between 0.2 and 0.5.

A final simple regularization technique that is often used is *early stopping*. Early stopping, as the name suggests, refers to stopping the training process before its pre-scheduled termination. This is done by defining a number of iterations in which the validation loss has to decrease to continue training. If the validation loss does not decrease in the said number of iterations, we discontinue the training and are left with the pre-terminated model.

3.1.4 Considerations

One very important aspect of Bayesian hyperparameter optimization is restricting the configuration space, as explained in Section 2.4.4. Bayesian optimization acts as a solution to limit the number of times we have to train the neural network, but we also want to limit the configuration space such that the surrogate model better can approximate the objective function. This can, to some degree, be achieved by selecting appropriate distributions from which to draw hyperparameter configurations.

Consider e.g. one neural network trained with learning rate $\psi_1 = 10^{-4}$ and another trained with learning rate $\psi_2 = 10^{-1}$. Adding a fixed change of $\psi_1 = 10^{-3}$ would have almost no effect for the model trained with ψ_2, but a huge effect for the model trained with ψ_1. This is because the learning rate has a multiplicative effect on the gradient descent, as seen in Eq. (2.1). The same applies to the weight regularization parameter, η, as shown in Eq. (3.1). Thus, one would not prefer to sample learning rates and regularization parameters from a uniform distribution, but instead, use a logarithmic uniform distribution to accommodate this multiplicative effect.

Another example is the number of layers and layer width. A change observed from a model 1 hidden layer to a model with 2 hidden layers can be quite significant. But the change from 128 hidden layers to 129 is not so much. The same logic applies to

the layer width as well. Thus, popular choices for these hyperparameters are powers of 2.

Another consideration to take into account is hyperparameters that we apriori know do not often work well together. Combining batch normalization and dropout is a combination that often leads to higher loss and slower convergence. Thus, it is recommended to only use one of them. This is because the random dropout of neurons introduces noise into the statistics used to perform batch normalization, which might have unwanted effects. On the other hand, some hyperparameters have a strong relation, such as the use of MSE as the cost function with the use of a $\mathcal{L}_2$ regularization term. Combining these two is equal to maximizing the posterior distribution of the likelihood function of a linear regression model with a Gaussian prior distribution on the weights (Goodfellow et al., 2016, Section 5.6)

In Table 3.1.1 suggestions for an appropriate configuration space for the hyperparameters mentioned earlier in this section are provided. Keep in mind that these are general suggestions, and domain knowledge from human experts should be valued when defining the overall hyperparameter configuration space.

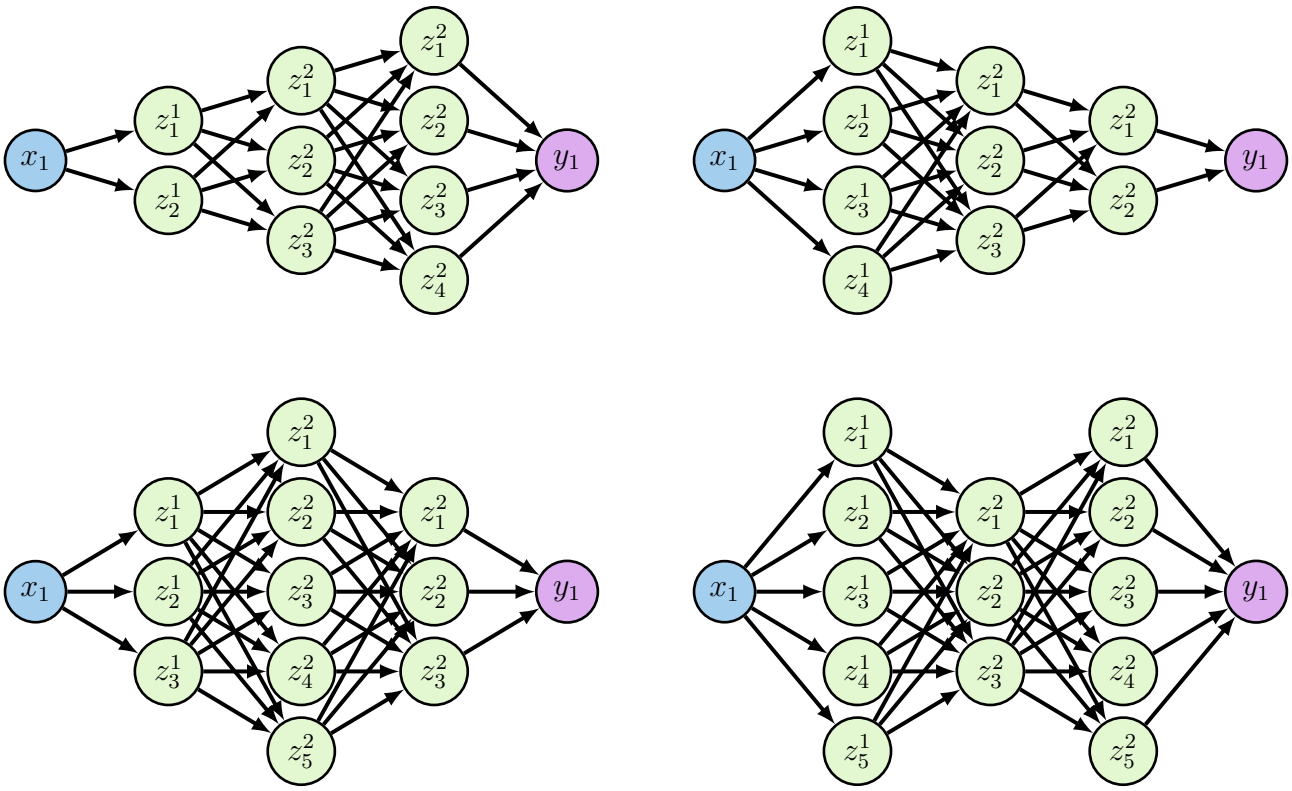


Figure 3.1.2: Illustration of multiple architecture styles for a deep feed-forward neural network

Table 3.1.1: Typical DFFNN hyperparameters. HP = Hyperparameters.

HP	Description	Configuration space
	Model architecture and Model complexity	
Number of layers	Number of hidden layers.	Powers of 2: {2, 4, 8, 16, ...}
Layer width	Number of neurons in each layer.	Powers of 2: {16, 32, 64, ...}
Activation function	How the neurons are activated.	Nonlinear
	Training and Optimizer	
Epochs	Number of times each training point is passed through the network.	
Batch-size	Number of training points passed through each backward propagation.	Powers of 2: {16, 32, 64, ...}
Learning rate	Step length in gradient descent.	Loguniform$[10^{-5}, 10^{-1}]$
Cost function	Which function used to calculate the loss between the models prediction and the actual expected output.	$\mathcal{L}_2$
Optimizer	Which optimizer to use.	Adaptive learning rate optimizers
Batch Normalization	Wether or not to normalise the input data in each batch.	Boolean
	Regularization	
Weight regularizaition	Penalization of models with large weights.	Loguniform$[10^{-6}, 10^{-1}]$
Dropout rate	Probability of deactivating neurons.	Uniform[0.2, 0.5]
Early stopping	Number of epochs to continue training without improving validation loss	

3.2 Software Review

There is currently a lot of software available for hyperparameter optimization, each with its own twist and facilitating machine learning models in different programming languages. Since the neural network which hyperparameters we wish to run Bayesian optimization on is implemented with PyTorch (Paszke et al., 2019) in Python, the first requirement for the software is that it must support Python. Some software also provides the option of using different surrogate models, but since we wish to use a Gaussian process regressor, the package must at least support this.

In addition to these two requirements, a set of evaluation points was considered when analysing the different software packages. Most importantly was the ease of use of the software. Since part of the primary objective of this work is to form a practical foundation, the entry-level to get started was considered more important than choosing the most efficient or lightweight software. Thus, the level of documentation and examples provided by the software provider was considered important. Furthermore, the flexibility of the software was considered. As mentioned earlier, many packages provide e.g. different surrogate models, but we also wish to e.g. be able to select different acquisition functions, kernels and easily control the exploitation-exploration trade-off. This is important in case we in the future decide we want to explore other aspects of BO and HPO. Ultimately, since part of the secondary objective of this work is to review software that Solution Seeker can use to perform HPO, the maintenance of the software was also considered an important aspect as they cannot rely on third-party software that might break.

The rest of this chapter gives a short review of different hyperparameter optimization software that meets the requirements given above.

Bayesian Optimization

Bayesian Optimization is a pure Python implementation of Bayesian global optimization with Gaussian processes (Nogueira, 2014). Its documentation is rather sparse and most of the framework's features has to be extracted from different examples. The software is furthermore not very flexible and does not for instance provide built-in features for conditional, integer or categorical configurations of hyperparameters. It also only provides a method for maximizing, although changing our minimization problem to a maximization problem can easily be achieved. In addition, there are no direct PyTorch integration examples to get started. The software is however easy to use, and serves as a great introduction to BO with GPR, but based on the last commit dates (Dec 19, 2020, Jul 13, 2020, as of Nov 18 2021) the package is not very often maintained.

SciKit Optimize

SciKit Optimize (Head et al., 2021) implements several methods for SMBO, one of them being Bayesian optimization with GPs. It is well enough documented with

an intuitive user guide, API reference guide and multiple examples. The software is a bit more flexible than Bayesian Optimization, with e.g. support for integer and categorical search spaces, as well as some other optimization functions such as random search. SciKit Optimize does not provide any direct PyTorch integration or examples, but with the documentation, it is easy enough to combine the two. A new version was released in October 2021, and based on commit logs it looks rather well maintained.

Optuna

Optuna provides a lightweight, versatile, and platform-agnostic architecture (Akiba et al., 2019) written entirely in Python with few dependencies. It is highly flexible with support for many optimization functions, as well as integration for many other software packages such as e.g. SciKit Optimize or BoTorch. It also scales well. The package is furthermore greatly documented with PyTorch examples and is well maintained.

Ax

Ax, developed by Facebook, is another platform that provides a lightweight and flexible interface for optimization and provides many out-of-the-box Bayesian optimization methods among others. It is easy to use, and the package is well documented with several examples, including using it with PyTorch. The software is also very well maintained, with several recent releases.

BoTorch

BoTorch is a library for Bayesian optimization built on top of PyTorch that provides great modularity and is best used in tandem with Ax (Balandat et al., 2020). Its Gaussian process regressor is built on GPyTorch (Gardner et al., 2018). Even though the software is well documented, if you are not familiar with Ax and Bayesian optimization, BoTorch recommend starting with Ax for an easy-to-use suite. It is also very well maintained and adapts quickly to PyTorch updates.

Ray Tune

Tune is a Python library for experiment execution and hyperparameter tuning at any scale (Liaw et al., 2018). Rather than implementing their own search algorithms, they provide wrappers for open-source optimization packages such as Ax, Optuna, SciKit Optimize, and many others. This makes Tune extremely flexible. Using Tune's wrappers is very easy, and each wrapper comes with its own example and documentation. Tune's own features are also very well documented. Furthermore, Tune integrates great with PyTorch, and they provide several examples to get started. It is also being maintained well with several recent releases.

4 | Case Study

4.1 Case description

This case is designed to fulfil the primary objective of this exploratory work. The master thesis mentioned concerns hyperparameter optimization for a deep neural network VFM for oil & gas production. A simplified flow model of the oil production process can be seen in Fig. 4.1.1. Understanding the principles behind the VFM (Grimstad et al., 2021) and the oil & gas production process is not needed to understand this case or experiment (Section 4.2). Hence, no further explanation to these concepts are given.

The case reflects the thesis problem on a smaller scale. It involves performing HPO on a DFFNN on a smaller, but representative, data set for flow rate prediction in oil and gas production. The DFFNN is in this case used to estimate the total flow rate in a period where the installed multiphase flow meter was defective.

The data set used to create the neural network is sampled from a real-life implementation of the production process illustrated in Fig. 4.1.1. The total flow rate that is to be predicted is a continuous real-valued output. Thus, the problem at hand is a regression problem. The data set consists of seven types of measurements described in Table 4.1.1. The first six are used as input features, and the final, QTOT, as an output label. The data set contains 3099 samples, where each sample contains one measurement of each of the seven types.

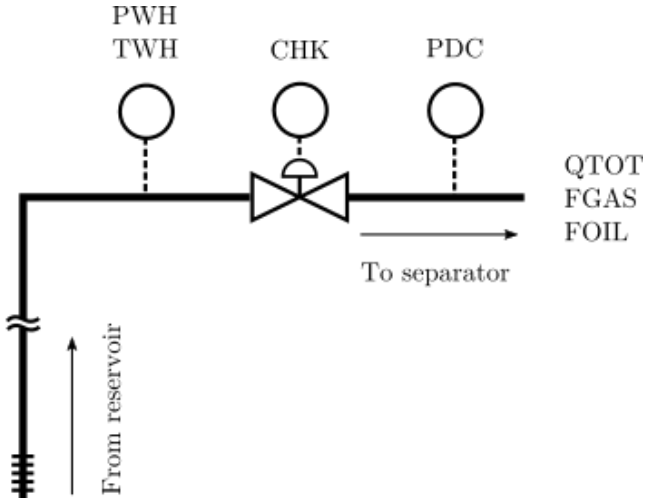


Figure 4.1.1: Simplified Oil & Gas production flow model. Oil, gas, and water is pumped up from a reservoir, goes through a choke valve that controls the flow, and is led into a separator that separates the three materials from each other. Figure from Grimstad (2020). For legend description, see Table 4.1.1.

Table 4.1.1: Case data set description

Legend	Description	Type	Range
TWH	Temperature upstream the choke	Feature	Scaled between $[0, 1]$
PWH	Preassure upstream the choke	Feature	Scaled between $[0, 1]$
CHK	Choke opening	Feature	$[0, 1]$, 0 being closed and 1 being fully opened
PDC	Preassure downstream the choke	Feature	Scaled between $[0, 1]$
FGAS	Fraction of gas to total flow	Feature	$[0, 1]$
FOIL	Fraction of oil to total flow	Feature	$[0, 1]$
QTOT	Total flow rate	Label	$\mathbb{R}$

4.2 Experiment design

This experiment was designed to answer the research questions stated in Section 1.3. Bayesian optimization is an optimization algorithm that aims to outperform the naive grid and random search algorithms described in Section 2.2.1. Furthermore, since RS is proven to outperform grid search if given the same computational budget (Section 2.2.1), RS was selected as the naive comparison ground.

4.2.1 Hypothesis

Random search allows for searching over more of the important hyperparameters than grid search. The same logic can somewhat be adapted to compare RS and BO as well. When optimizing $N = 1$ hyperparameters, the surrogate model cannot model any correlation between different hyperparameters (since there is only one). Thus, the Bayesian optimization model is not able to make an informative assumption on which hyperparameter affects the validation loss the most and focus on finding an appropriate value for this parameter. The advantage gained by modelling the black-box function from hyperparameter to validation loss is therefore expected to be less when the configuration space is small, and get bigger as the size of the configuration space increases. From this, the hypothesis for the experiment was formed:

> **Hypothesis:**
> Bayesian optimization should at least perform as well as random search for any size of configuration space, and when the configuration space reaches a critical size, Bayesian optimization should significantly outperform random search.

4.2.2 Method

Data pre-processing

First, a set of 500 consecutive samples was extracted from the data set to a test set to simulate the period with defective measurements. Then 260 randomly chosen

points of the remaining data set were extracted to a validation set. The remaining 2399 samples then formed the training set. The split into the three sets is illustrated in Fig. 4.2.1. No standardization or normalization was applied to the input of the data set, as all input values were in the range between 0 and 1.

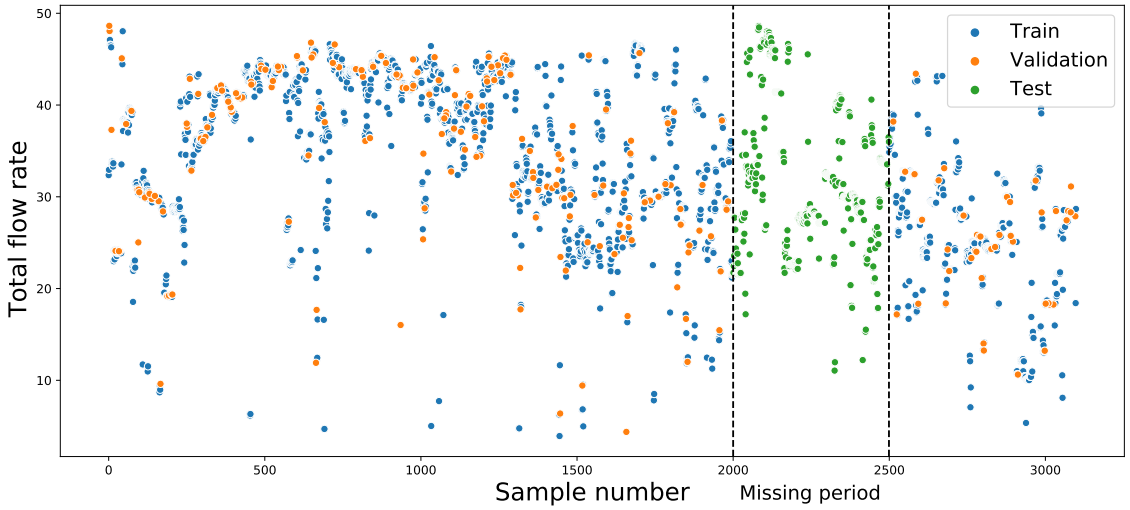


Figure 4.2.1: Case data set split into training, validation and test set.

Optimization process

This process applies to both the random search and Bayesian optimization.

It was decided to run five different experiments with five different configuration space sizes to investigate the effect the size of the configuration space had. Each experiment existed of 15 individual optimizations, where each optimization was given a budget of 100 trials. Put in other words, for each experiment Algorithm 2 was run 15 individual times, where T in step 2 was set to 100. The hyperparameters for the first five trials were selected at random such that the surrogate model had some observed data before any acquisition was made. For RS, it simply means that for each experiment, 15 times 100 random hyperparameters within the configuration space were tried.

Validation loss, step 4 of Algorithm 2, was then reported after the final epoch of each trial. This applies to RS as well. Reporting the validation loss from the best epoch of each trial was also considered. This was not chosen since the general idea is that longer training should lead to better performance. By reporting the validation loss from the best epoch, and thus possibly an early epoch, the BO would not be able to take full advantage of the training loop since the samples provided to the surrogate model is possibly terminated early. The network's state was then stored after each trial was completed.

After every experiment was done, each trial was further trained on the validation set with the same configuration from the corresponding trial. Then the validation-set-trained model of each trial was evaluated on the test set, and test error was reported. Pseudo-code for the experiment is given in Algorithm 3

The selection of covariance function is one of two important decisions that have to be made when using a GPR as described in Section 2.4.3. Williams and Rasmussen (2006), which many of the other works referred to base its knowledge on as well, deemed the squared exponential kernel (Eq. (2.8)) to be too smooth. Thus, a Matérn kernel with $\nu = 5/2$ (Eq. (2.9b)) was chosen as the covariance function for the Gaussian process regressor. The Matérn kernel with $\nu = 3/2$ (Eq. (2.9a)) could have been chosen as well.

The other important choice is the selection of the acquisition function. There exist no general acquisition function that repeatedly outperforms the others. Since the experiment was designed to investigate the role of the size of the configuration space some exploration was desired in the acquisition function, and purely exploitation acquisition functions, such as PI (Eq. (2.10)), were ruled unfit. Expected improvement (Eq. (2.11)) was thus chosen as the acquisition function. GP LCB (Eq. (2.12)) could have been chosen as well. The trade-off parameter ω was chosen to slightly explore more than to exploit. The value of ω is given in Section 4.3 as it was software dependant.

Computational budget

Giving a comparable budget to the Bayesian optimization and random search was important to be able to make a meaningful interpretation of the results. One factor that had to be taken into account was therefore the parallelizability of the optimization methods. This is because the parallel capacity of a computer is highly dependent on the available hardware. RS is out-of-the-box parallelizable as explained in Section 2.2.1, while BO is out-of-the-box sequential. The optimization was therefore restricted to run on 1 CPU without any multithreading to make a fair comparison between the methods that are not dependent on the parallel capacity of the computer. Thus, giving the two methods the same amount of trials gives them the same computational budget. The budget was set to 100 trials as mentioned earlier.

Neural Network Model

The neural network which hyperparameters' were optimized was implemented in PyTorch (Gardner et al., 2018) and was adapted from Grimstad (2020). The model used only fully connected linear layers. Every neurone in the input layer and the hidden layers used ReLU activation. There was no activation used in the output layer. All model parameters were initialized using He-weight-initialization, which is designed for ReLU activation (He et al., 2015). MSE was used as the cost function, since predicting the flow rate is a regression problem. Furthermore, a $\mathcal{L}_2$ penalty term was added to the cost function for a regularizing effect. Finally, dropout was added after every hidden layer. Adam (Kingma and Ba, 2017) was chosen as the gradient descent optimizer, with a mini-batch size of 64 and training for 100 epochs. All inputs in the data set were used as features in the input layer of the neural network.

Hyperparameter- and Configuration Space- Selection

There exists as discussed earlier many hyperparameters, and selecting an appropriate configuration space is key to extracting the most out of the Bayesian optimization. This also applies to RS, as it too would be unwise to draw random samples from apriori highly likely unfit distributions (such as sampling a random learning rate between 10 and 1000). Furthermore, restricting the search space is also important for the GPR to be able to model the black-box function.

Learning rate, being probably the most important hyperparameter, was selected as the first hyperparameter to optimize. Then, experimenting with different complexities to investigate the impact on the model's capacity was desired. Thus, the number of layers and the number of hidden layers were selected as hyperparameter two and three. Finally, experimenting with the regularization of the network to avoid overfitting was desired. Thus, the dropout rate and the $\mathcal{L}_2$ penalty were selected as hyperparameters four and five. The distribution, domain and fixed value for these hyperparameters are given in Table 4.2.1.

The size of the domains for the chosen hyperparameters were chosen such that the combination of number of experiments and number of trials for each experiment could be executed in a reasonable amount of time on the available hardware, and still represent a thorough optimization process. This aspect also affected the domain of number of layers and layer width, as wider and deeper nets are more expensive to train than shallow and slim nets. The domain of the dropout rate extends to 0 to avoid a conditional configuration space. The fixed dropout rate was set to 0.2, as this was the lower recommended bound and the possible width of the nets in this case was not that wide. The other fixed values were set around the middle of the corresponding domain.

Five configuration spaces were selected, following the same logic as the selection of hyperparameters. The resulting configuration spaces are given in Table 4.2.2. When a hyperparameter was not included in the configuration space, it was given the fixed value given in Table 4.2.1.

Table 4.2.1: Hyperparameter selection

Hyperparameter	Distribution	Domain	Fixed value
Learning rate	Logarithmic Uniform	$[10^{-5}, 10^{-1}]$	N/A
Number of layers	Integer Uniform	{2, 4, 6, ..., 10}	6
Layer width	Integer Uniform	{30, 45, 60, ..., 120}	70
Dropout rate	Uniform	[0.0, 0.5]	0.2
$\mathcal{L}_2$ penalty	Logarithmic Uniform	$[10^{-6}, 10^{-1}]$	10^{-3}

Table 4.2.2: Case configuration space selections

Name	Hyperparameters
CS1	Learning rate
CS2	Learning rate, and number of layers
CS3	Learning rate, number of layers, and layer width
CS4	Learning rate, number of layers, layer width, and dropout rate
CS5	Learning rate, number of layers, layer width, dropout rate, and $\mathcal{L}_2$ penalty

Algorithm 3 Pseudo-code for experiment

$\mathcal{H} \leftarrow \emptyset$
$\mathcal{M} \leftarrow \emptyset$
for $k \leftarrow 1$ **to** 15 **do**
    $\mathcal{H}^k \leftarrow \emptyset$
    $\mathcal{M}^k \leftarrow \emptyset$
    for $t \leftarrow 1$ **to** 100 **do**
        $\boldsymbol{\lambda}^* \leftarrow \text{argmin}_\lambda \, \alpha_{EI}(\boldsymbol{\lambda}, P(f|\mathcal{H}^k_{t-1}))$
        Train $\mathcal{A}_{\boldsymbol{\lambda}^*}$ and evaluate $\boldsymbol{V}(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train}, D_{valid})$
        $\mathcal{H}^k \leftarrow \mathcal{H}^k \cup (\boldsymbol{\lambda}^*, \boldsymbol{V}(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train}, D_{valid})$
        $\mathcal{M}^k \leftarrow \mathcal{M}^k \cup \mathcal{A}_{\boldsymbol{\lambda}^*}$
        Update posterior distribution $P(f|\mathcal{H}^k) \propto P(f) \times P(\mathcal{H}^k|f)$
    end for
end for
$\mathcal{E} \leftarrow \emptyset$
for $k \leftarrow 1$ **to** 15 **do**
    for $t \leftarrow 1$ **to** 100 **do**
        Train $\mathcal{M}^k_t$ on D_{valid}
        Evaluate test error $\boldsymbol{V}(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train \cup valid}, D_{test})$
        $\mathcal{E} \leftarrow \mathcal{E} \cup \boldsymbol{V}(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}^*}, D_{train \cup valid}, D_{test})$
    end for
end for
return $\mathcal{E}$

4.3 Implementation

Ray Tune was selected as the desired Python library to implement the experiment. Since Tune provides wrappers for many other different hyperparameter optimization methods, learning to use Tune seemed like a clever choice for further works. Furthermore, adapting the neural network to work with Tune was straightforward.

Tune does not provide a direct method for BO with GPR, but it is possible to achieve this by using some of the many search algorithms that Tune provides. What this work considered the simplest way to achieve this was to use the *BayesOptSearch* that provides a wrapper for the Bayesian Optimization package (Nogueira, 2014). But as described in Section 3.2 this library is not very well maintained nor flexible. Thus, using the *SkoptSearch* wrapper for SciKit Optimize's *Optimize* object seemed a good choice. However, the configuration space had to be written somewhat different from the standard Tune way. Thus, the choice fell on using the *OptunaSearch* wrapper for Optuna's *Samplers*, which again provides an interface called *SkoptSampler* for SciKit Optimize's *Optimize* class. Although it at first glance might seem a little intricate, it was quite straightforward to implement as illustrated in Listing 1. The trade-off parameter was set to 0.05 (xi in Listing 1). 0.01 is the standard value, where higher values favour more exploration, and lower values favour more exploitation. The Matérn kernel with $\nu = 5/2$ is the standard kernel used by the *Optimizer* class. The class also continuously normalizes the history of previous samples ($\mathcal{H}$) each time a new sample is taken. See Eriksen (2021) for a complete GitHub repository with the implementation of the experiment.

```
from ray.tune.suggest.optuna import OptunaSearch
from optuna.integration import SkoptSampler

sampler = SkoptSampler(
    skopt_kwargs={
        "base_estimator": "GP",
        "n_initial_points": 5,
        "acq_func": "EI",
        "acq_func_kwargs":{
            "xi": 0.05
        }
    }
)

algorithm = OptunaSearch(sampler=sampler)
```

Listing 1: Python code for defining Bayesian optimization with Gaussian process regression in Ray Tune using the *OptunaSearch* and *SkoptSampler* interfaces.

5 | Results and Discussion

5.1 Results

The results from the experiment are given in Table 5.1.1 and Fig. 5.1.1 - 5.1.6. Only test errors, E, that improve from the previous trial were saved, such that at trial T, the reported test error is the lowest value obtained to and including T:

$$E'_T = \min(E_{1:T-1}\,,\, E_T)$$

An illustration of the improvement from trial 1 to 100 of each experiment and a box plot of the individual experiments is given in Fig. 5.1.2 - 5.1.6. For the improvement plots (Fig. 5.1.2a - 5.1.6a), the mean is shown with high opacity, and all individual trials are shown with low opacity and matching colours. Early trials that reach outside an illustrative meaningful scope are left out. For the box plots (Fig. 5.1.2b - 5.1.6b) the whiskers extend to points that lie within 1.5 IQRs of the lower and upper quartile.

Please note that the plots in Fig. 5.1.2 - 5.1.6 are mainly intended to be used to compare the difference between BO and RS within the corresponding CS. Therefore, the y-axis in the plots have different ranges. A comparison between the changes in different configuration spaces can be seen in Fig. 5.1.1.

Table 5.1.1: Experiment results after 100 trials.
CS = Configuration Space. RS = Random Search. BO = Bayesian Optimization. STD = Standard Deviation. Δ = Change [%]. Test error is measured in MSE.

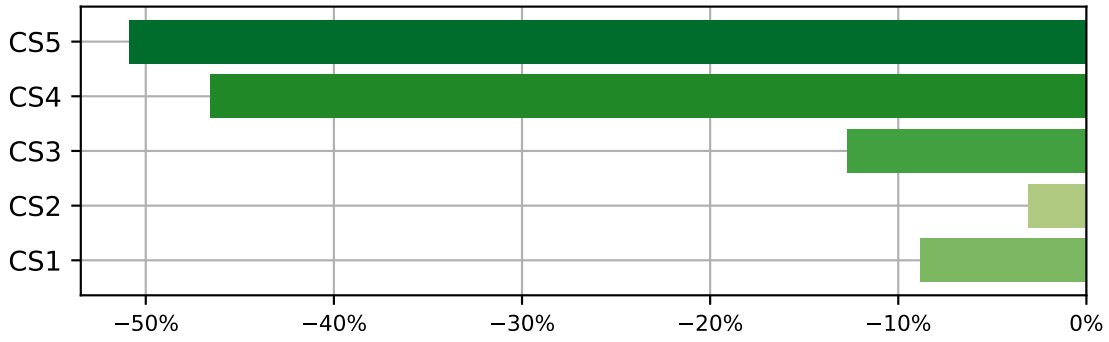
	Mean Test Error			STD of Test Error			Best Test Error	
CS	RS	BO	Δ [%]	RS	BO	Δ [%]	RS	BO
CS1	32.880	29.973	-8.84	4.808	6.606	+37.39	25.581	16.517
CS2	28.910	28.022	-3.07	5.673	7.078	+24.78	17.495	15.362
CS3	31.096	27.146	-12.70	8.544	6.601	-22.75	18.030	15.721
CS4	2.280	1.218	-46.58	0.968	0.257	-73.42	1.011	0.833
CS5	2.675	1.313	-50.91	0.762	0.271	-64.47	1.495	0.960

The results show that BO outperformed RS on average in every experiment. Optimizing the second-largest configuration space, CS4, resulted in the lowest mean for both methods. Optimizing the smallest configuration space, CS1, resulted in the highest mean also for both methods. The most significant improvement is observed with the largest configuration space, CS5, with a 50.91% lower mean test error. The least significant improvement is observed with the second smallest configuration space, CS2, only decreasing the test error by 3.07%.

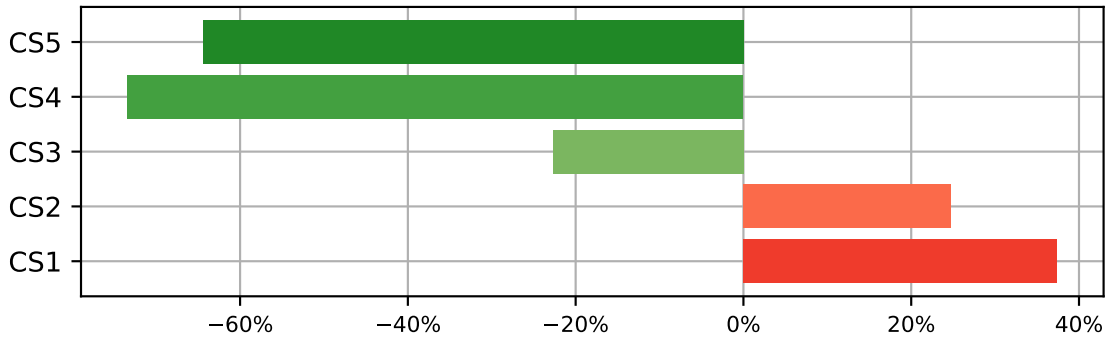
Bayesian optimization resulted in a lower standard deviation for the three largest configuration spaces, while the STD was bigger for the two smallest. All percentage-wise changes in STD were quite significant, with the best improvement found when optimizing CS4, and the worst decrease found when optimizing CS1. BO obtained the lowest standard deviation from experiment CS4, whereas RS obtained it from CS5. Bayesian optimization obtained the biggest standard deviation when optimizing CS2, and random search from optimizing CS3.

Both methods obtained their lowest test error when optimizing CS4, and the highest from CS1. The best test error obtained in each experiment was lower for Bayesian optimization in all cases.

The trial improvement plots in Fig. 5.1.2a - 5.1.6a shows that the mean of the BO is lower than the mean of the RS throughout the trials for all experiments, except for the first iterations of CS2 and CS3 in Fig. 5.1.3a and, Fig. 5.1.4a respectively.

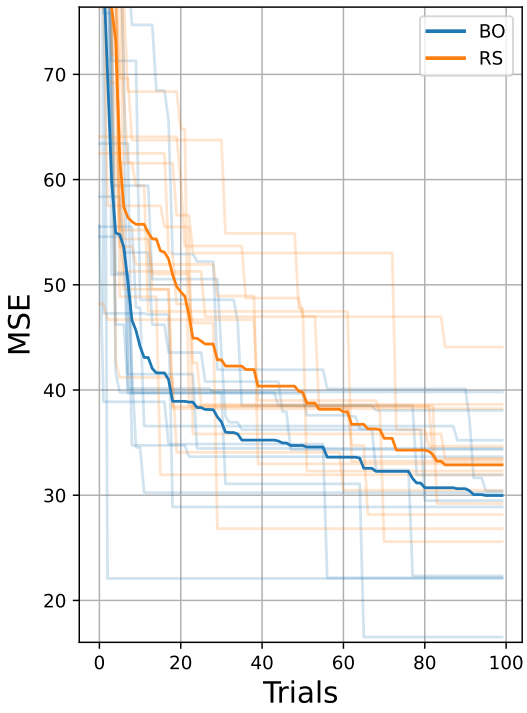


(a) percentage-wise change of mean test error.

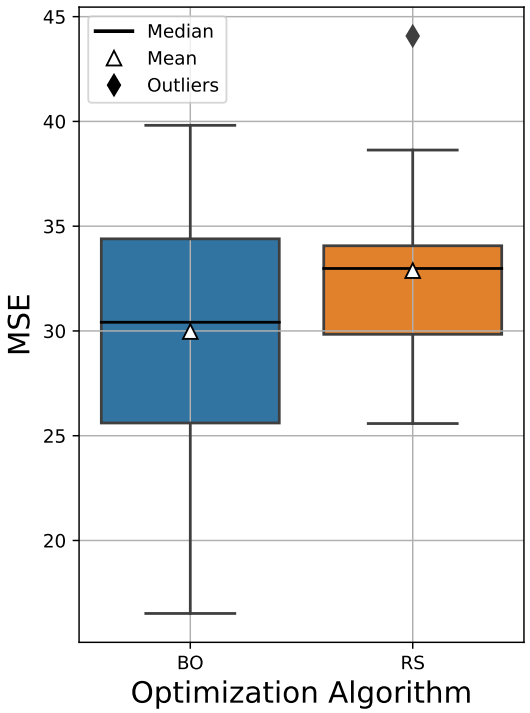


(b) percentage-wise change of test error STD.

Figure 5.1.1: Change from RS to BO after 100 trials

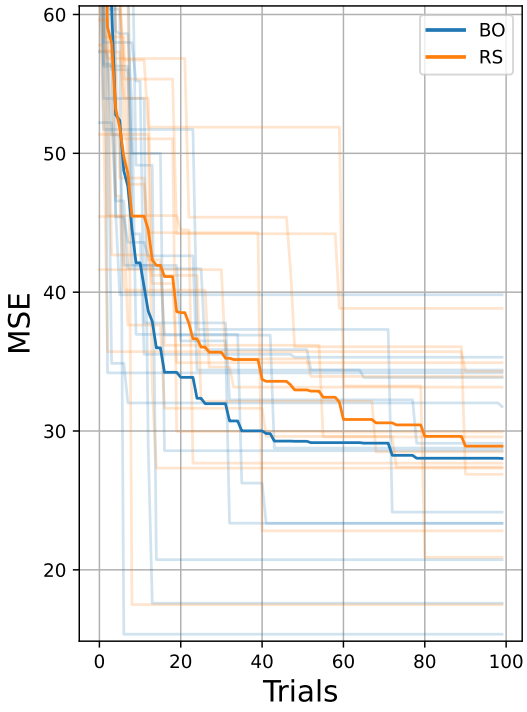


(a) Improvements from trial three to 100. BO outperforms RS from the first acquisition.

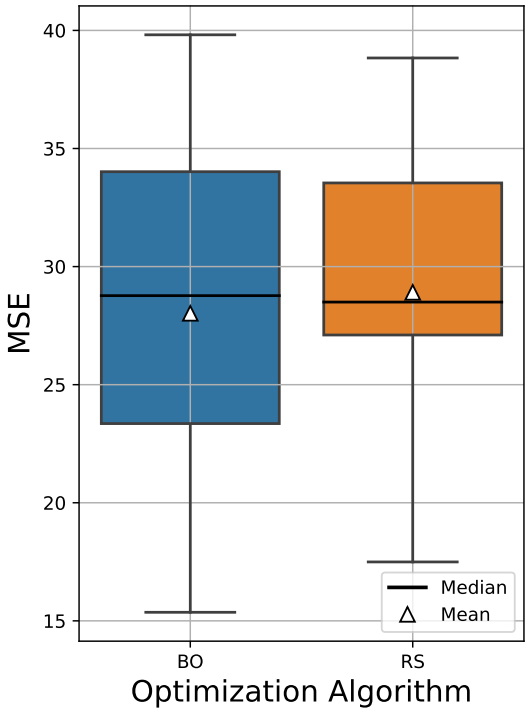


(b) Results after 100 trials. BO finds the best result, and its median and mean are lower.

Figure 5.1.2: Results from optimizing CS1.

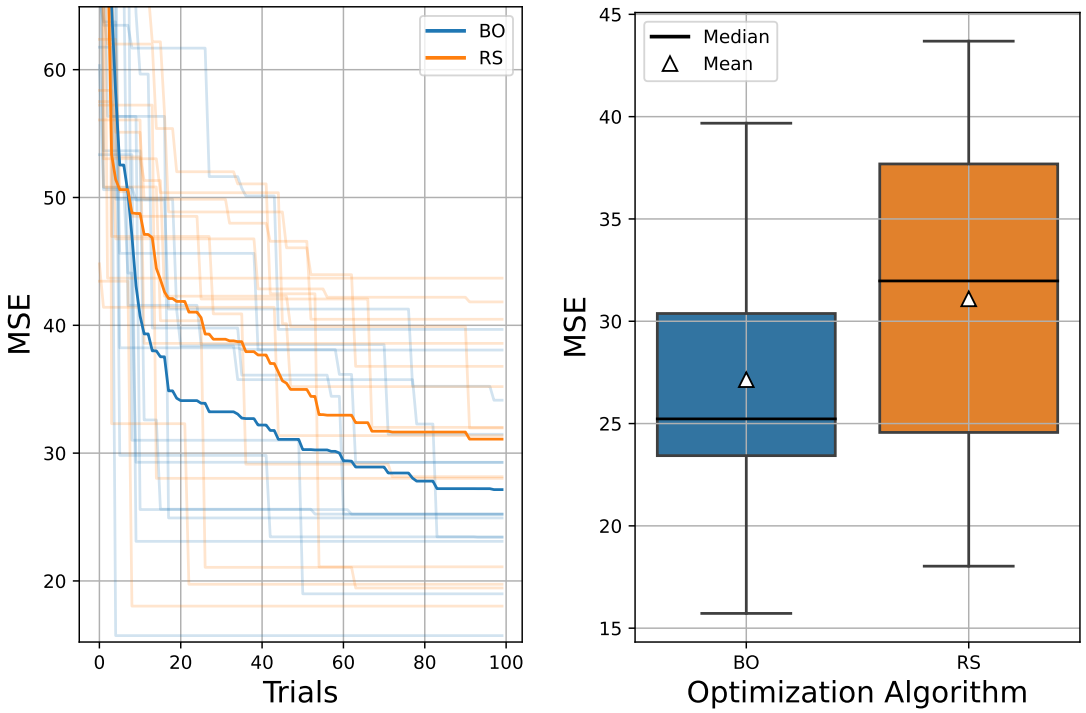


(a) Improvements from trial three to 100. BO outperforms RS from the second acquisition.



(b) Results after 100 trials. BO finds the best and worst result. BO's mean is lower but median higher.

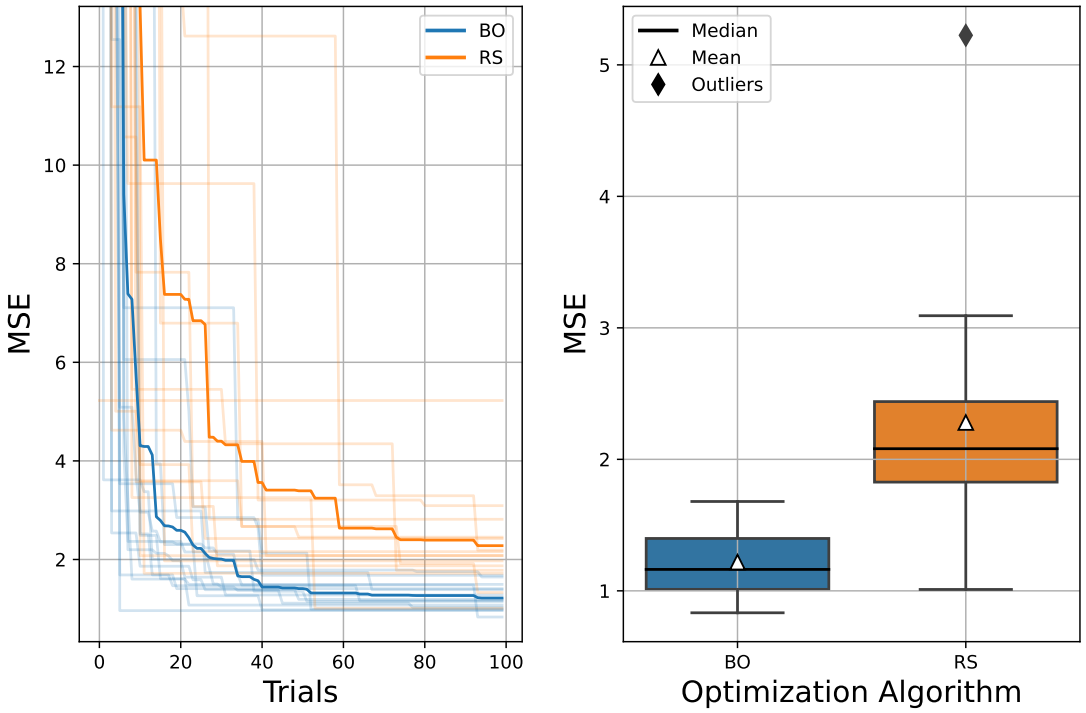
Figure 5.1.3: Results from optimizing CS2.



(a) Improvements from trial three to 100. BO outperforms RS from the fourth acquisition.

(b) Results after 100 trials. BO finds the best result, and its median and mean are lower.

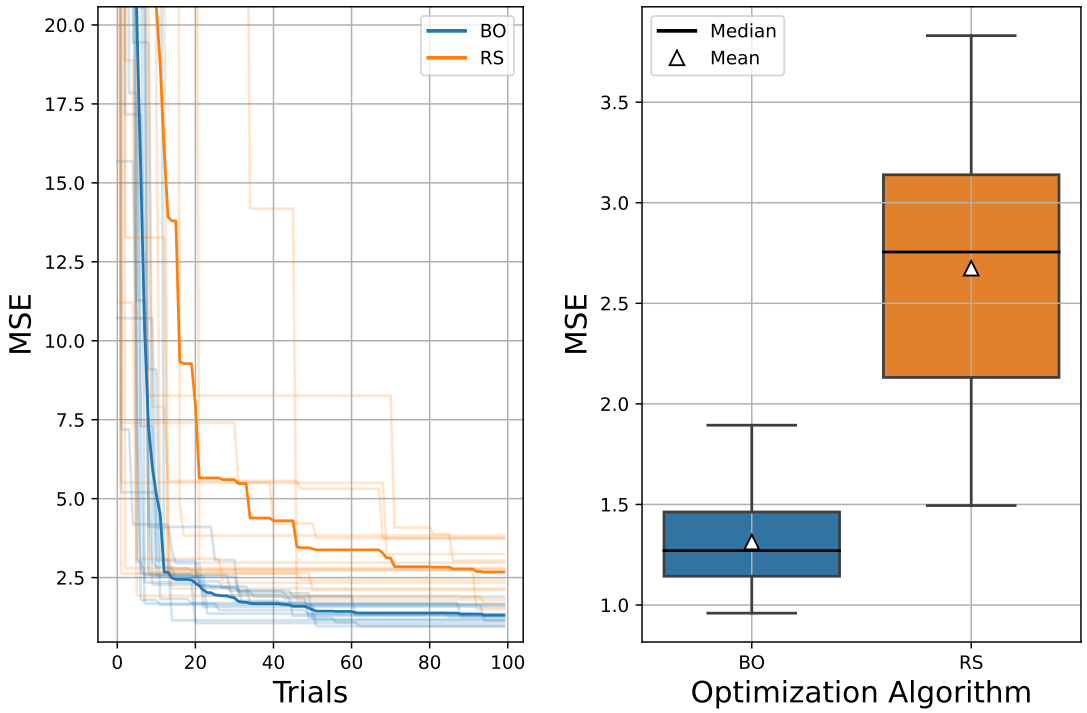
Figure 5.1.4: Results from optimizing CS3.



(a) Improvements from trial 10 to 100. BO outperforms RS from the first acquisition.

(b) Results after 100 trials. BO finds the best result, and its median and mean are lower.

Figure 5.1.5: Results from optimizing CS4.



(a) Improvements from trial 10 to 100. BO outperforms RS from the first acquisition.

(b) Results after 100 trials. BO finds the best result, and its median and mean are lower.

Figure 5.1.6: Results from optimizing CS5.

5.2 Discussion

It should be noted that although some significance can be given to the results, to generalize any of the points discussed in this section to the performance of any Bayesian optimization method for a general deep feed-forward neural network would be wrong. Any generalization at all would require many more experiments.

5.2.1 A note about high means and standard deviations

The first thing to notice is the high mean obtained in CS1, CS2 and CS3. This is likely because the dropout rate was set fixed at 0.2, while the number of epochs was kept at 100 and the maximum possible width was quite shallow. Thus, it is highly likely that the network was underfitting.

Training the many models that the use of dropout effectively results in could possibly have resulted in a lower test error mean if the number of epochs and/or layer width had been increased. Dropping random nodes in the net without allowing the net to train for sufficient epochs does not allow the training error to be minimized enough, and the corresponding test error is effected thereafter. Increasing the layer width reduces the impact that dropping nodes have on the representational capacity of the network, and the underfitting likely observed could have been reduced.

The use of dropout can also explain the large standard deviations from CS1, CS2 and CS3. The randomness in the dropout causes a somewhat more random learning behaviour, which with more epochs or wider layers could have been evened out.

Although more training or more complex networks could have improved the observed errors, it is highly likely that the fixed dropout rate was set too high, causing too much regularization. This theory is also supported by the resulting best hyperparameter configurations from both Bayesian optimization and random search on CS4 and CS5 given in Table A.7 - A.10. Twenty-eight out of the total 30 individual BO experiments turned dropout completely off, and the largest dropout rate used was 0.057, which effectively is zero. In addition, the largest dropout value from the best trials out of the 30 RS experiments was 0.043, which again effectively is zero. This highlights the importance of choosing an appropriate configuration space in accordance with the neural network model.

5.2.2 Performance

Even if the high test errors from CS1-3 can be explained, this makes it difficult to compare the absolute values of the test error of CS1, CS2 and CS3 with the absolute values of CS4 and CS5. However, the experiment showed that BO outperformed RS on average for every configuration space, which supports the first part of the hypothesis.

Comparing the percentage-wise change of the means between the methods can provide some insight into the importance of the configuration space. One possibility is that the large difference between the percentage-wise change of CS1-3 and CS4-5 is also due to the fixed dropout rate. More likely, it is because SMBO-algorithms are tailored for expensive objective functions. The small configuration space in CS1, CS2, and CS3 allow RS to try out enough combinations of hyperparameters to obtain a comparable result to BO. With the growth of the configuration space, the amount of random possible configurations simply become too many, while the surrogate model of the BO can guide the optimization towards a lower validation error. This supports the second part of the hypothesis.

Comparing the change in STD provide further insight into the importance of the configuration space. Again, it is possible that the differences of observed change in STD is also due to the fixed dropout rate. However, the STD improves for CS3, which has a fixed dropout rate. Thus, the decrease of STD in CS1 and CS2 could be because the acquisition function favoured too much exploration, causing no convergence towards any minimum. Another possibility is that it favoured too much exploitation. The surrogate model is too sure of its posterior distribution because of the small configuration space, causing the optimization to converge towards different local minima. Nevertheless, the observed changes in STD also supports the second part of the hypothesis.

The drop of BO mean from CS4 to CS5 might be an indication of the problem of scalability with GP (Section 2.4.4) within a computational budget. This theory is also supported by the lower standard deviation also obtained in CS4. With the in-

troduction of an extra hyperparameter, the CS becomes too large for the surrogate to model the black-box function well enough within the computational budget. The acquisition function is therefore not able to find an as optimal set of hyperparameters as in CS4 within the allocated number of trials. This does not contradict the second part of the hypothesis, but do illustrate that when the configuration space becomes too large, the BO with GP might not be able to significantly outperform RS. However, the differences between the BO mean and STD of CS4 and CS5 is not that large, and more experiments are required if this result was to be generalized. The convergence plots in Fig. 5.1.2a - 5.1.6a illustrates that BO performs at least as well as RS within the computational budget. The gap between the mean test error obtained by the two methods seems to be the largest around the quarter mark of the allowed budget, and somewhat shrink towards the end. This is a clear indication of the GP being able to fit the objective function early, which allows for smart acquisitions in the early trials. The slight favour of exploration might have affected the results a bit, and maybe a lower test error could have been obtained with standard settings of the acquisition function or even favour more exploitation.

It was stated in Section 4.2.2 that no standardization was applied to the data set. Applying this would probably have led to lower test errors for both RS and BO. Furthermore, since the validation set was quite small, the use of *k-fold cross-validation* could probably have improved the BO process, but would not affect the RS since no validation was applied to RS. But, if a model was to be selected from the experiment, instead of investigating the development of test error, this model would have had to be selected based on the validation error to avoid data leakage as explained in Section 2.1.1. In that case, the use of cross-validation probably would have improved RS as well.

6 | Conclusion and further work

In this work, the theory of hyperparameter optimization using Bayesian optimization with Gaussian processes was explored. A case study with a hypothesis was designed and performed to gain practical experience with HPO and to investigate the applicability of BO with GP on Solution Seeker's VFM. The results from the case study support the hypothesis. They indicate that BO is able to outperform a naive hyperparameter optimization technique on a deep neural network, as the mean test error of all experiments were lower for BO than RS. The results furthermore show that BO can perform as well as – or better than – RS within the given computational budget, as illustrated by Fig. 5.1.2a - 5.1.6a. They also indicate the importance of the size of the configuration space. The percentage-wise change of both mean and STD show that BO is able to significantly outperform RS when the configuration space grows.

The high absolute values of both mean and STD obtained from the configuration spaces with a fixed dropout rate highlights the importance of the careful selection of hyperparameters and configuration space design to extract the most of the BO. The percentage-wise decrease of STD in CS1 and CS2 also highlights this importance.

Again, no general conclusion can be drawn from the results of this case study.

For future work, the following tasks are suggested:

- Due to the promising results from the case study, a large scale HPO using BO with GP should be performed on Solution Seeker's VFM to extract further performance from their current design. The use of early stopping with Ray Tune's *Trial Schedulers* should be investigated to speed up the optimization process.
- Since scalability is one of the biggest challenges with Gaussian processes, future work should investigate promising methods to overcome this issue. In particular, the use of Tree-structured Parzen Estimators (Bergstra, Bardenet et al., 2011; Hutter et al., 2011), the use of deep neural networks (Snoek, Rippel et al., 2015) and Hyperband (Li et al., 2017; Falkner et al., 2018) would be interesting to look into. TPEs are particularly interesting due to their native support of different hyperparameter domains.
- As no acquisition function has been shown to be better than the other, investigating the possibility of using a portfolio of acquisition functions instead of a single function could lead to better results when optimizing a larger neural network.

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*.

Balandat, Maximilian et al. (2020). 'BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization'. In: *Advances in Neural Information Processing Systems 33*. URL: http://arxiv.org/abs/1910.06403.

Bellman, Richard E (2015). *Adaptive control processes*. Princeton university press.

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 (2013). 'Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures'. In: *International conference on machine learning*. PMLR, pp. 115–123.

BP (2021). *Statistical Review of World Energy*. https://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy/primary-energy.html. Accessed: 2021-12-07.

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*.

Brownlee, Jason (2019). *A Tour of Machine Learning Algorithms*. Accessed: 2021-11-19. URL: https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/.

Eriksen, K. V. (2021). *Hyperparameter Tuning with Ray Tune and PyTorch*. https://github.com/knuteriksen/project.

Falcone, Gioia (2009). 'Multiphase Flow Metering Principles'. In: *Developments in petroleum science* 54, pp. 33–45.

Falkner, Stefan, Aaron Klein and Frank Hutter (2018). 'BOHB: Robust and efficient hyperparameter optimization at scale'. In: *International Conference on Machine Learning*. PMLR, pp. 1437–1446.

Feurer, Matthias and Frank Hutter (2019). 'Hyperparameter optimization'. In: *Automated machine learning*. Springer, Cham, pp. 3–33.

Gardner, Jacob R et al. (2018). 'Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration'. In: *arXiv preprint arXiv:1809.11165*.

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.

Grimstad, Bjarne et al. (Aug. 2021). 'Bayesian neural networks for virtual flow metering: An empirical study'. In: *Applied Soft Computing* 112, p. 107776. DOI: 10.1016/j.asoc.2021.107776.

Guyon, Isabelle et al. (2015). 'Design of the 2015 chalearn automl challenge'. In: *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1–8.

He, Kaiming et al. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv: 1502.01852 [cs.CV].

Head, Tim et al. (Oct. 2021). *scikit-optimize/scikit-optimize*. Version v0.9.0. DOI: 10.5281/zenodo.5565057. URL: https://doi.org/10.5281/zenodo.5565057.

Hoffman, Matthew, Eric Brochu, Nando de Freitas et al. (2011). 'Portfolio Allocation for Bayesian Optimization.' In: *UAI*. Citeseer, pp. 327–336.

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.

Jones, Donald R (2001). 'A taxonomy of global optimization methods based on response surfaces'. In: *Journal of global optimization* 21.4, pp. 345–383.

Jordan, Jeremy (2018). *Setting the learning rate of your neural network*. Accessed: 2021-11-26. URL: https://www.jeremyjordan.me/nn-learning-rate/.

Kevin P., Murphy (2012). *Machine Learning : A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. The MIT Press. ISBN: 9780262018029. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=480968&site=ehost-live.

Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Kohavi, Ron and George H John (1995). 'Automatic parameter selection by minimizing estimated error'. In: *Machine Learning Proceedings 1995*. Elsevier, pp. 304–312.

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*.

Mendoza, Hector et al. (2016). 'Towards automatically-tuned neural networks'. In: *Workshop on Automatic Machine Learning*. PMLR, pp. 58–65.

Nogueira, Fernando (2014). *Bayesian Optimization: Open source constrained global optimization tool for Python*. Accessed: 2021-11-02. URL: https://github.com/fmfn/BayesianOptimization.

Nwankpa, Chigozie et al. (2018). *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. arXiv: 1811.03378 [cs.LG].

Orduz, Juan (2019). *An Introduction to Gaussian Process Regression*. Accessed: 2021-11-02. URL: https://juanitorduz.github.io/gaussian_process_reg/.

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.

Ruder, Sebastian (2017). *An overview of gradient descent optimization algorithms*. arXiv: 1609.04747 [cs.LG].

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.

Snoek, Jasper, Oren Rippel et al. (2015). 'Scalable bayesian optimization using deep neural networks'. In: *International conference on machine learning*. PMLR, pp. 2171–2180.

Solution Seeker (2021). *OUR STORY*. https://www.solutionseeker.no/our-story. Accessed: 2021-12-11.

Srinivas, Niranjan et al. (2009). 'Gaussian process optimization in the bandit setting: No regret and experimental design'. In: *arXiv preprint arXiv:0912.3995*.

Srivastava, Nitish et al. (2014). 'Dropout: a simple way to prevent neural networks from overfitting'. In: *The journal of machine learning research* 15.1, pp. 1929–1958.

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.

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

Appendix

A Hyperparameter configuration results

Table A.1: Resulting hyperparameters from running BO on CS1

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.038978	6	70	0.2	0.001
1	0.025517	6	70	0.2	0.001
2	0.014969	6	70	0.2	0.001
3	0.043859	6	70	0.2	0.001
4	0.033565	6	70	0.2	0.001
5	0.042346	6	70	0.2	0.001
6	0.012247	6	70	0.2	0.001
7	0.019587	6	70	0.2	0.001
8	0.026204	6	70	0.2	0.001
9	0.032201	6	70	0.2	0.001
10	0.015655	6	70	0.2	0.001
11	0.011580	6	70	0.2	0.001
12	0.027853	6	70	0.2	0.001
13	0.024381	6	70	0.2	0.001
14	0.008643	6	70	0.2	0.001

Table A.2: Resulting hyperparameters from running RS on CS1

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.061086	6	70	0.2	0.001
1	0.028097	6	70	0.2	0.001
2	0.029203	6	70	0.2	0.001
3	0.058032	6	70	0.2	0.001
4	0.008964	6	70	0.2	0.001
5	0.031935	6	70	0.2	0.001
6	0.039364	6	70	0.2	0.001
7	0.019722	6	70	0.2	0.001
8	0.031454	6	70	0.2	0.001
9	0.015146	6	70	0.2	0.001
10	0.071717	6	70	0.2	0.001
11	0.005265	6	70	0.2	0.001
12	0.054283	6	70	0.2	0.001
13	0.005527	6	70	0.2	0.001
14	0.015418	6	70	0.2	0.001

Table A.3: Resulting hyperparameters from running BO on CS2

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.045386	4	75	0.2	0.001
1	0.033335	6	75	0.2	0.001
2	0.005805	2	75	0.2	0.001
3	0.019245	8	75	0.2	0.001
4	0.066054	4	75	0.2	0.001
5	0.100000	2	75	0.2	0.001
6	0.058739	2	75	0.2	0.001
7	0.100000	2	75	0.2	0.001
8	0.007749	10	75	0.2	0.001
9	0.037568	2	75	0.2	0.001
10	0.008333	2	75	0.2	0.001
11	0.007840	2	75	0.2	0.001
12	0.008795	10	75	0.2	0.001
13	0.016330	2	75	0.2	0.001
14	0.007161	10	75	0.2	0.001

Table A.4: Resulting hyperparameters from running RS on CS2

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.053756	4	75	0.2	0.001
1	0.009623	10	75	0.2	0.001
2	0.020373	4	75	0.2	0.001
3	0.007051	10	75	0.2	0.001
4	0.031699	4	75	0.2	0.001
5	0.023822	8	75	0.2	0.001
6	0.050998	2	75	0.2	0.001
7	0.019950	8	75	0.2	0.001
8	0.076682	4	75	0.2	0.001
9	0.008646	10	75	0.2	0.001
10	0.011217	8	75	0.2	0.001
11	0.043114	4	75	0.2	0.001
12	0.067415	4	75	0.2	0.001
13	0.079420	2	75	0.2	0.001
14	0.062705	4	75	0.2	0.001

Table A.5: Resulting hyperparameters from running BO on CS3

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.017771	2	120	0.2	0.001
1	0.004969	2	120	0.2	0.001
2	0.020326	2	120	0.2	0.001
3	0.026351	4	120	0.2	0.001
4	0.005332	2	120	0.2	0.001
5	0.006563	2	60	0.2	0.001
6	0.017186	8	120	0.2	0.001
7	0.009667	4	120	0.2	0.001
8	0.023909	2	30	0.2	0.001
9	0.100000	2	75	0.2	0.001
10	0.045931	6	75	0.2	0.001
11	0.037802	4	120	0.2	0.001
12	0.100000	2	120	0.2	0.001
13	0.085129	2	105	0.2	0.001
14	0.090691	2	60	0.2	0.001

Table A.6: Resulting hyperparameters from running RS on CS3

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.036954	4	90	0.2	0.001
1	0.038285	4	60	0.2	0.001
2	0.013764	2	75	0.2	0.001
3	0.078412	4	120	0.2	0.001
4	0.035482	4	90	0.2	0.001
5	0.009841	6	105	0.2	0.001
6	0.029415	8	90	0.2	0.001
7	0.018969	2	60	0.2	0.001
8	0.035863	4	105	0.2	0.001
9	0.003153	2	60	0.2	0.001
10	0.049789	2	75	0.2	0.001
11	0.059467	4	90	0.2	0.001
12	0.013808	4	120	0.2	0.001
13	0.027011	2	120	0.2	0.001
14	0.020445	2	75	0.2	0.001

Table A.7: Resulting hyperparameters from running BO on CS4

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.010697	8	30	0.000	0.001
1	0.007768	4	75	0.000	0.001
2	0.100000	2	120	0.000	0.001
3	0.008419	6	120	0.000	0.001
4	0.002887	8	75	0.000	0.001
5	0.023138	4	30	0.000	0.001
6	0.003935	8	120	0.012	0.001
7	0.000396	6	120	0.000	0.001
8	0.000422	10	120	0.000	0.001
9	0.001449	10	120	0.000	0.001
10	0.008395	6	105	0.000	0.001
11	0.006439	4	45	0.000	0.001
12	0.002115	8	120	0.000	0.001
13	0.042398	2	30	0.057	0.001
14	0.037370	2	30	0.000	0.001

Table A.8: Resulting hyperparameters from running RS on CS4

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.004058	2	105	0.024	0.001
1	0.007495	4	45	0.034	0.001
2	0.000620	8	75	0.000	0.001
3	0.003422	4	75	0.013	0.001
4	0.000330	4	105	0.028	0.001
5	0.000274	4	75	0.001	0.001
6	0.000075	8	105	0.001	0.001
7	0.000105	4	90	0.023	0.001
8	0.002494	4	60	0.011	0.001
9	0.009181	10	90	0.021	0.001
10	0.004834	8	105	0.002	0.001
11	0.010717	2	60	0.001	0.001
12	0.020375	4	105	0.010	0.001
13	0.007880	8	90	0.000	0.001
14	0.000560	4	60	0.006	0.001

Table A.9: Resulting hyperparameters from running BO on CS5

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.002221	8	60	0.000	0.000002
1	0.011948	8	120	0.000	0.100000
2	0.002712	4	120	0.000	0.039834
3	0.000701	2	120	0.000	0.001644
4	0.005740	6	60	0.000	0.100000
5	0.032642	2	30	0.000	0.000162
6	0.100000	4	120	0.000	0.000001
7	0.009823	8	30	0.000	0.100000
8	0.005612	4	90	0.000	0.000014
9	0.000841	4	45	0.000	0.000023
10	0.012865	4	120	0.000	0.000001
11	0.100000	2	120	0.000	0.000001
12	0.073330	4	45	0.000	0.000001
13	0.003785	6	105	0.000	0.000001
14	0.008872	8	30	0.000	0.100000

Table A.10: Resulting hyperparameters from running RS on CS5

	Learning rate	Hidden layers	Layer width	Dropout rate	$\mathcal{L}_2$ penalty
0	0.030713	4	90	0.023	0.001856
1	0.033123	4	60	0.043	0.000010
2	0.006402	10	105	0.005	0.005162
3	0.000023	10	60	0.002	0.019217
4	0.000311	6	105	0.010	0.000344
5	0.000045	4	90	0.004	0.000119
6	0.000073	4	105	0.002	0.000001
7	0.000506	8	75	0.007	0.069848
8	0.008674	8	75	0.001	0.000002
9	0.029060	2	90	0.017	0.000001
10	0.085448	6	105	0.004	0.000265
11	0.000395	6	90	0.006	0.000005
12	0.000472	6	75	0.000	0.019245
13	0.000546	8	45	0.003	0.001854
14	0.022547	4	120	0.026	0.000031