BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GAUSSIAN PROCESSES BASED HYPER-OPTIMALIZATION OF NEURAL NETWORKS

HYPER-OPTIMALIZACE NEURONOVÝCH SÍTÍ ZALOŽENÁ NA GAUSSOVSKÝCH PROCESECH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MARTIN COUFAL**
AUTOR PRÁCE

SUPERVISOR **Ing. KAREL BENEŠ**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) | Academic year 2019/2020

Master's Thesis Specification

Student: **Coufal Martin, Bc.**
Programme: Information Technology Field of study: Intelligent Systems
Title: **Gaussian Processes Based Hyper-Optimization of Neural Networks**
Category: Artificial Intelligence
Assignment:

1. Get acquainted with Gaussian processes for regression
2. Get acquainted with neural networks, focusing on common hyper-parameters
3. Implement an optimizer based on Gaussian processes
4. Demonstrate its performance on a suitable set of analytical functions
5. Design and implement a toolkit for hyper-parameter search for neural network training
6. Demonstrate its performance on a suitable task

Recommended literature:

- https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/46180.pdf

Requirements for the semestral defence:

- Items 1, 2, 3, and development of item 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Beneš Karel, Ing.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 20, 2020
Approval date: November 1, 2019

Abstract

The goal of this thesis is to create a lightweight toolkit for artificial neural network hyper-parameter optimisation. The optimisation toolkit has to be able to optimise multiple, possibly correlated hyper-parameters. I solved this problem by creating an optimiser that uses Gaussian processes to predict the influence of the hyper-parameters on the resulting neural network accuracy. Based on the experiments on multiple benchmark functions, the toolkit is able to provide better results than random search optimisation and thus reduce the number of necessary optimisation steps. The random search optimisation provided better results only in the first few optimisation steps before Gaussian process optimisation creates sufficient model of the problem. However the experiments on MNIST dataset show that random optimisation achieves almost always better results than used GP optimiser. These differences between the experiments results are probably caused by insufficient complexity of the benchmarks or by selected parameters of the implemented optimiser.

Abstrakt

Cílem této diplomové práce je vytvoření nástroje pro optimalizaci hyper-parametrů umělých neuronových sítí. Tento nástroj musí být schopen optimalizovat více hyper-parametrů, které mohou být navíc i korelovány. Tento problém jsem vyřešil implmentací optimalizátoru, který využívá Gaussovské procesy k predikci vlivu jednotlivých hyperparametrů na výslednou přesnost neuronové sítě. Z provedených experimentů na několika benchmark funkcích jsem zjistil, že implementovaný nástroj je schopen dosáhnout lepších výsledků než optimalizátory založené na náhodném prohledávání a snížit tak v průměru počet potřebných kroků optimalizace. Optimalizace založená na náhodném prohledávání dosáhla lepších výsledků pouze v prvních krocích optimalizace, než si optimalizátor založený na Gaussovských procesech vytvoří dostatečně přesný model problému. Nicméně téměř všechny experimenty provedené na datasetu MNIST prokázaly lepší výsledky optimalizátoru založeného na náhodném prohledávání. Tyto rozdíly v provedených experimentech jsou pravděpodobně dány složitostí zvolených benchmark funkcí nebo zvolenými parametry implementovaného optimalizátoru.

Keywords

hyper-parameter tuning, Gaussian processes, neural networks optimisation, regression problem solving, kernels

Klíčová slova

optimalizace hyper-parametrů, Gaussovské procesy, optimalizace neuronových sítí, řešení regresních problémů, kernely

Reference

COUFAL, Martin. *Gaussian Processes Based Hyper-Optimalization of Neural Networks.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Karel Beneš

Rozšířený abstrakt

Neuronová síť je výpočetní model inspirovaný biologickým neuronem lidské nervové soustavy. Tento model je používán pro řešení problémů spojených se strojovým učením, jako jsou například zpracování řeči, detekce podvodů nebo medicínská diagnostika.

Neuronová síť se skládá z jednotlivých vrstev umělých neuronů, kde každý neuron má vstupní vektor $\mathbf{x} = (x_1, ..., x_n)^T$, vektor vah $\mathbf{w} = (w_1, ..., w_n)^T$, bias Θ, aktivační funkci a výstup. Výstup neuronu y je dán jeho aktivační funkcí f: $y = f(\mathbf{x}^T\mathbf{w} + \Theta)$. Tyto umělé neurony jsou ve vrstvách propojeny tak, že výstup jedné vrstvy je propojen se vstupem vrstvy následující. První vrstva neuronové sítě je pak označována jako vstupní vrstva, poslední vrstva je označována jako výstupní vrstva a všechny ostatní vrstvy jsou označovány jako skryté.

Aby tyto modely mohly správně fungovat, je nutné je "naučit" na dostatečném množství trénovacích dat. Toto učení je řízeno pomocí učícího algoritmu a funguje na principu aktualizace parametrů neuronových sítí – vektorů vah a biasů. Učící algoritmus trénuje neuronovou síť postupně, na množinách trénovacích dat kterým se se říká *dávky*. Tyto trénovací data jsou nejdříve přivedena na vstup neuronové sítě a dále propagována neuronovou sítí. Výstup sítě je následně porovnán s cílovým vektorem daných trénovacích dat pomocí chybové (*loss*) funkce. Výstup tého funkce se označuje jako *loss* nebo chyba sítě. Nakonec jsou hodnoty parametrů neuronové sítě aktualizovány pomocí zvoleného učícího algoritmu. Míra o kolik jsou parametry neuronové sítě upraveny je řízena *koeficientem učení*. Celý proces učení je pak za účelem zlepšení přesnosti modelu možno opakovat, přičemž jeden cyklus učení přes všechna trénovací data se nazývá *epocha*.

Výsledná přesnost neuronové sítě je ovlivněna mnoha faktory, jako jsou struktura neuronové sítě, učící algoritmus, použitá trénovací data a inicializace parametrů neuronové sítě. Faktorům které ovlivňují strukturu nebo způsob učení neuronové sítě se říká hyperparametry a jejich nastavení výrazně ovlivňuje výslednou přesnost neuronové sítě. Běžné hyper-parametry ovlivňující strukturu neuronové sítě jsou například počet skrytých vrstev, počet neuronů ve skrytých vrstvách a aktivační funkce, zatímco běžné hyper-parametry ovlivňující učení sítě jsou koeficient učení, velikost dávek nebo počet epoch.

K optimalizaci hyper-parametrů se běžně využívá manuální ladění nebo optimalizátory založené na náhodném prohledávání. Existují však i složitější přístupy, jako Bayesovské optimalizační metody nebo jiné optimalizační algoritmy spojené se strojovým učením. Základní struktura Bayesovských optimalizátorů je popsána pomocí formalismu *Sequential Model-Based Optimisation* (SMBO), který definuje hlavní smyčku cyklu optimalizace. V prvním kroku se vytvoří model rozdělení pravděodobnosti $p(y|\mathbf{x}, \mathcal{D})$, kde $\mathbf{x}$ je množina testovacích dat a $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_i, y_i)\}$ je množina trénovacích dat – množina nastavení hyper-parametrů $\mathbf{x}_i$ pro které je již chyba sítě y_i známa. Výběr testovacích dat je řízen pomocí *Domain Search Strategy* (DSS). V dalším kroku vybere *akviziční funkce* (AF) pomocí vytvořeného pravděpodobnostního modelu $p(y|\mathbf{x}, \mathcal{D})$ následující nastavení hyperparametrů, které je následně použito pro trénování neuronové sítě. Nakonec se dvojice $(\mathbf{x}_i, y_i)$ přidá do množiny trénovacích dat $\mathcal{D}$, kde y_i je chyba sítě při nastavení hyperparametrů $\mathbf{x}_i$.

Pro reprezentaci modelu rozdělení pravděpodobnosti se běžně používá Gaussovský proces, náhodný les, nebo Tree-Parzen Estimation. V této práci jsou pro tento účel využity Gaussovské procesy. Zatímco klasické rozdělení pravděpodovnosti náhodné proměnné popisuje vlastnosti skaláru nebo vektoru, stochastický proces popisuje funkce. Stochastický proces $y(\mathbf{x})$ je definován sdruženým rozdělením pravděpodobnosti pro každou konečnou množinu hodnot $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_D)$. Gaussovský proces je stochastický proces, kde pod-

míněné rozdělení pravděpodobnosti $p(y(\mathbf{x}_1), \ldots, y(\mathbf{x}_D)|\mathbf{x}_1, \ldots, \mathbf{x}_D)$ je D-rozměrné normální rozdělení. Toto vícerozměrné rozdělení pravděpodobnosti je popsáno jeho střední hodnotou μ a kovariační maticí Σ. Střední hodnota udává střed rozdělení pravděpodobnosti, zatímco hodnoty na diagonále kovariační matice udávají rozptyl pro každou dimenzi a zbytek hodnot definuje korelaci mezi každými dvěma náhodnými proměnnými. Hodnota korelace je dána *kernelem*, který definuje tvar tohoto pravděpodobnostního rozdělení. Pro účely hyper-optimalizace je prostor optimalizovaných hyper-parametrů v závislostí na chybové funkci modelován podmíněným rozdělením pravděpodobnosti $p(\mathbf{x}|\mathbf{y})$, kde $\mathbf{x} = (x_1, \ldots, x_n)^T$ je vektor náhodných proměnných reprezentujících testovací data a $\mathbf{y} = (y_1, \ldots, y_m)^T$ je vektor náhodných proměnných reprezentujících trénovací data..

Implementovaný toolkit je navžen jako knihovna v programovacím jazyce Python, která využívá popsaný SMBO formalismus s pravděpodobnostním modelem založeným na Gaussovských procesech za účelem optimalizace hyper-parametrů neuronových sítí. Toolkit je rozdělen do dvou hlavních balíčků `optimiser` a `nnbridge`. Balíček `optimiser` obsahuje implementaci Gaussovského procesu, optimalizátorů, DSS, AF a kernelů. Toolkit obsahuje tři optimalizátory: *Grid optimiser*, *Random optimiser* a *GP optimiser*, založené na prohledávání na mřížce, náhodném prohledávání a Gaussovských procesech. Implementace DSS obsahuje dvě strategie, *grid* a *random*, které vybírají testovací data z domény na základě prohledávání na mřížce nebo pomocí náhodného prohledávání. Toolkit dále obsahuje implementaci tří akvizičních funkcí, z nichž dvě jsou založeny na výběru na základě nejnižší odhadované funkční hodnoty (*Lower Confidence Bound*) a třetí vybírá následující nastavení hyper-parametrů dle očekávaného zlepšení (*Expected Improvement*). Dále `optimiser` obsahuje implementaci pěti kernelů (*konstantní*, *lineární*, *RBF*, *Laplaceovský* a *Matérn*) a dvou metod pro jejich skládání. Balíček `nnbridge` se stará o korektní propojení optimalizátoru s neuronovou sítí. Parametry tohoto propojení, optimalizované hyper-parametry a rozsah domén, na kterých jsou hyperparametry optimalizovány, je nutné definovat pomocí konfiguračního souboru. Toolkit dále poskytuje skript `wrapper.py`, který slouží jako rozhraní pro spuštění optimalizace z příkazové řádky. Implementovaný optimalizátor dále umožňuje automatickou optimalizaci parametrů kernelu a neurčitosti v modelu GP, která je založena na maximalizaci *log likelihood* funkce GP modelu.

Implementovaný toolkit byl otestován na dvou různých benchmark funkcích až v pěti dimenzích a také na neuronové síti využívající dataset MNIST. První set experimentů na benchmark funkcích byl zaměřen na porovnání GP optimalizátoru s optimalizátory založenými na náhodném prohledávání a prohledávání na mřížce, zatímco další experimenty byly zaměřeny na porovnání implementovaných kernelů, DSS, AF a automatické optimalizace parametrů GP optimalizátoru. Experimenty na benchmark funkcích prokázaly, že optimalizátor založený na Gaussovských processech je schopen dosáhnout lepších výsledků než optimalizátor založený na náhodném prohledávání. Experimenty dále ukázaly, že random DSS je lepší strategie pro výběr testovacích dat než grid DSS a navíc dosahuje lepších výsledků při menším množství testovacích dat, což urychluje výpočetní rychlost modelu. Dále bylo ukázáno, že na zvolených benchmark funkcích dosahuje nejlepších výsledků AF Expected Improvement, jejíž použití vede k výraznému zlepšení již v prvních krocích optimalizace. Nejlepších výsledků mezi testovanými kernely dosáhl optimalizátor s Laplaceovským kernelem. Poslední část experimentů věnovaná automatické optimalizaci ukázala, že optimalizace neurčitosti zároveň s parametry kernelu může vést k výraznému zhoršení přesnosti modelu. Na druhou stranu automatická optimalizace výhradně parametrů kernelu vedla k dosažení lepších výsledků ve většině experimentů. Při experimentech na neuronové síti bylo dosaženo nejlepších výsledků při prohledávání na mřížce a náhodném

prohledávání, v závislosti na množině optimalizovaných hyper-parametrů. GP optimalizace dosáhla o něco horších výsledků, pravděpodobně v důsledku zvolených parametrů GP optimalizátoru.

Gaussian Processes Based Hyper-Optimalization of Neural Networks

Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of Ing. Karel Beneš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Martin Coufal
June 3, 2020

Acknowledgements

I would like to thank my thesis supervisor Ing. Karel Beneš for professional advice on the researched topic, helpful notes regarding the text of this thesis and tips about the implementation details of the optimisation toolkit.

Contents

1 Introduction **2**

2 Neural Networks and their Hyper-parameters **3**
2.1 Structure of Neural Networks 3
2.2 Learning in Neural Networks 4
2.3 Hyper-parameters of Neural Networks 6

3 Hyper-parameter Optimisation **12**
3.1 Methods of Hyper-parameters Tuning 12
3.2 Gaussian Process 17
3.3 Using Gaussian Processes for Regression 19
3.4 Parametrization of GP optimiser 22

4 Design and Implementation of Toolkit for Hyper-optimisation of Neural Networks **28**
4.1 Toolkit Design 28
4.2 Toolkit Implementation 30
4.3 Usage Examples 37
4.4 GPOP Customisation 40

5 Experiments **42**
5.1 Benchmarks for Hyper-parameter Optimisation 42
5.2 Testing the Toolkit on Benchmarks 43
5.3 Neural Networks Experiments 49

6 Conclusion **54**

Bibliography **55**

Chapter 1

Introduction

Neural networks are computational models used to tackle machine learning tasks such as clustering, classification, regression, density modelling or data denoising. Specifically, this includes problems such as image and voice recognition, fraud detection, machine diagnostics, medical diagnoses and process modelling and control. For these models to work, they need to be trained on a sufficient amount of training data. The resulting efficiency and accuracy depend on many factors, such as the type and structure of the used neural network, learning algorithm, used training data and network parameter initialisation. The parameters that define the structure or learning aspects of the neural network are called hyper-parameters and their effect is often correlated.

The impact of hyper-parameters settings on resulting accuracy of the neural network is quite substantial and since the training of the network can take a noticeable amount of time, it is beneficial to find acceptable hyper-parameters in as few training rounds as possible. This thesis concerns with hyper-parameter optimisation of neural networks, design and implementation of the Gaussian process based hyper-optimisation toolkit and its comparison with several methods that are commonly used in hyper-parameter optimisation to demonstrate its efficiency.

There are many approaches that aim to solve the hyper-parameter optimisation problem. Manual tuning of hyper-parameters is still widely used, but to be effective, it requires a lot of user's knowledge about the optimised neural network and might be quite time consuming. Automated optimising solutions, on the other hand, do not require as much user's time and knowledge at the expense of computing time. Those solutions might use more straightforward approaches as grid search and random search, or more complex solutions such as Bayesian or Evolutionary optimisation.

This thesis is divided into several chapters as follows: the second chapter contains a description of neural networks focusing on structural and learning aspects influenced by its hyper-parameters settings. The third chapter lists common hyper-parameter optimisation methods, including a detailed description of Gaussian processes based optimisation. Chapter 4 provides an elaborated description of the design and implementation of the toolkit for hyper-optimisation along with possible alterations that can be done by the end user to customise the optimizer to fit specific neural network. The overview of performed experiments with implemented toolkit and evaluation of its efficiency can be found in Chapter 5.

Chapter 2

Neural Networks and their Hyper-parameters

Artificial neural networks (NN) are computational models loosely inspired by biological neural networks. They consist of subsequent layers, where each layer is composed of individual artificial neurons. The actual architecture of each network and its learning process can vary since neural networks is a broad term involving a lot of different neural network types and learning algorithms that can be applied to various problems. But the basic idea behind all neural networks is the same: based on some training data or feedback from the environment, the learning algorithm updates the neural network's parameters in order to solve the given problem.

Parameters in NN are coefficients of the model itself and can be estimated or learned from data. Concretely, parameters are usually input weights and biases of the individual neurons. Hyper-parameters, on the other hand, influence the structure or learning process of the NN and need to be explicitly set before training of the network.

2.1 Structure of Neural Networks

The basic element of the NN is an artificial neuron, also called a unit or a node. It consists of a number of inputs $x_1 ... x_n$, usually represented by a vector $\mathbf{x} = (x_1, ..., x_n)^T$, input weights $\mathbf{w} = (w_1, ..., w_n)^T$, bias Θ, activation function $f(\cdot)$, and output y. The output of the artificial neuron is determined as follows: all input values are multiplied by corresponding weights, and together with bias form the input of the activation function. The activation function takes this input and computes the output value $y = f(\mathbf{x}^T\mathbf{w} + \Theta)$, as shown in Figure 2.1. For common activation functions, the power of a single neuron is limited to solving linearly separable problems.

More complex problems can be solved by creating layers of neurons, where each layer's output (except from last layer) is the input of the next subsequent layer. The NN consists of the input layer, number of hidden layers, and the output layer as shown in Figure 2.2. The neurons in the input layer correspond to network's input, while the output layer represents network's output. The NN has n hidden layers of neurons, where each layer can have different number of neurons. Connections between two layers can be between every two nodes (fully connected layer) or just between some subset of nodes (convolutional layer).

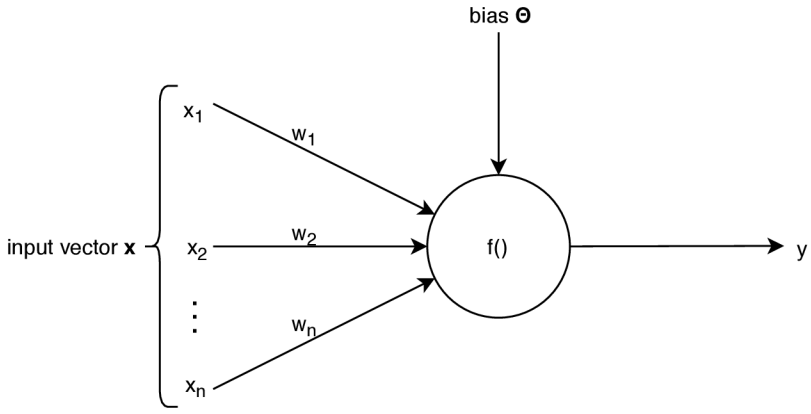


Figure 2.1: Basic structure of artificial neuron.

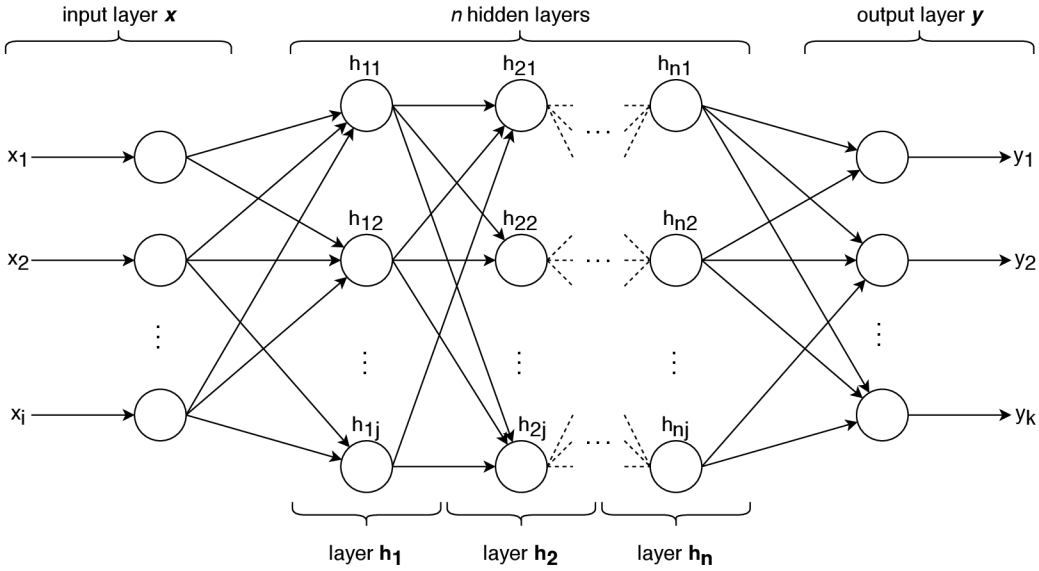


Figure 2.2: Basic structure of artificial neural network with i inputs, n hidden convolutional layers $\mathbf{h_1}, ..., \mathbf{h_n}$ (each consisting of j neurons) and output layer with k outputs.

The activation function, the number of hidden layers and the number of neurons in hidden layers are network's most common structural hyper-parameters. Features of these hyper-parameters and their influence on the NN are described in detail in Section 2.3.1.

2.2 Learning in Neural Networks

Learning in neural networks is a process of updating the parameters in order to achieve better accuracy. To evaluate this accuracy, network uses an error function and labelled training data. Labelled means each input vector $\mathbf{x}$ has a corresponding target vector $\mathbf{t}$, which denotes desired network's output. Then, a learning algorithm uses the error function $E(\cdot)$ to calculate the error and updates the parameters $\mathbf{w}$ of the NN. The exact principle of updating the parameters differs in each algorithm, but in order to introduce common hyper-parameters further described in Section 2.3.2, below is described the principle of training the NN using Gradient Descent algorithm [32].

Gradient Descent in an iterative algorithm that finds a local minimum by updating the weights by a small steps in the opposite direction of the error function's gradient ∇E:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}), \tag{2.1}$$

where τ is the current step and η is the learning rate. The learning rate is a hyper-parameter that controls how much are the weights changed in each step of the algorithm and its effect is further discussed in Section 2.3.2.

The gradient ∇E is a vector that points in a direction of fastest increase of the function. By updating the weights in the opposite direction of ∇E, Gradient Descent algorithm approximates to a local minimum as shown in Figure 2.3. To find better optimum, more initial settings of the weight $\mathbf{w}$ can be used, but there is no guarantee that global optimum will be found. Another way of getting out of the local optimum is using momentum hyper-parameter, which is described in the next section. Although, recent studies show that the local minima found by Gradient Descent algorithm in larger, multi-dimensional loss space of the NN are not such a problem, since their quality is comparable to the global minimum [10].

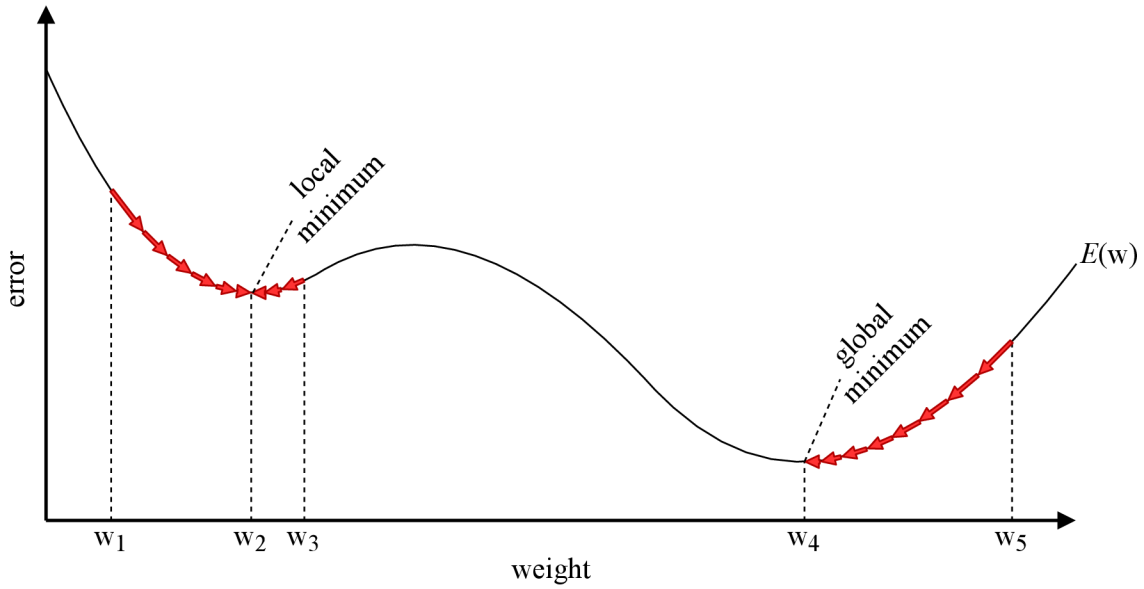


Figure 2.3: An example of weight optimisation using Gradient Descent algorithm. Figure shows the influence of a weight parameter w on the error function $E(\cdot)$, where the points w_1, w_3 and w_5 are different initial settings of weight w, resulting in different progress of Gradient Descent algorithm. The red arrows represent a change of the error when the current weight w^τ is updated to the value $w^{\tau+1}$. The weight update is in the opposite direction to a gradient of the error function $E(\cdot)$ in concrete settings of w^τ.

The gradient approximately equals to a derivative of the error function $E(\cdot)$ with respect to the weights $\mathbf{w}$ multiplied with how much are the weights changed [9]:

$$\nabla E \approx \frac{\partial E}{\partial \mathbf{w}} \nabla \mathbf{w} \tag{2.2}$$

Although the gradient can be calculated directly with respect to each weight individually, NNs usually have huge amount of parameters and therefore it's not very usable in

practice. Therefore, Backpropagation algorithm is commonly used. Backpropagation uses a local message passing scheme in which the information is sent alternately forwards and backwards through the network to efficiently evaluate gradient one layer at the time using the chain rule [23].

The error function $E(\cdot)$ denotes a measure that evaluates the difference between network's output and target vector and its selection depends solely on the features of the solved problem. Resulting output is called training or validation loss, depending on whether the input of the error function was training or validation data. One of the most common forms of the error function that is used in regression is sum of squares error function:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2 , \tag{2.3}$$

where $\mathbf{x}_n$ (for $n = 1, ..., N$) is a training set comprising of N input vectors, $\mathbf{y}(\mathbf{x}_n, \mathbf{w})$ is network's output vector for given input $\mathbf{x}_n$, $\mathbf{w}$ is weights vector and $\mathbf{t}_n$ is a target vector corresponding to n-th sample in training set.

The update of the parameters is done on a small sets of the training data, called *batches*. Depending on the size of those batches, Gradient Descent algorithms can be further divided into Stochastic Gradient Descent (SGD), Batch Gradient Descent and Mini-batch Gradient Descent, where SGD uses one training sample at a time, batch methods use whole training set at a time and Mini-batch methods use between one and all samples. One complete cycle through all the training data is called an *epoch* and can be repeated in order to increase the accuracy of the NN.

The number of epochs or the number of samples in one batch are an important hyper-parameters of the network, because suitable settings of these hyper-parameters can improve generalisation and prevent underfitting or overfitting of the NN. Underfitting problem is when the network is not able to make correct predictions on unseen data because it's too simple or isn't trained well enough. Overfitting occurs when the NN is too complex and over-adapted to the training data. That leads to loosing the ability of generalisation and therefore poor accuracy on the testing data. However in contrast to other hyper-parameters, a suitable number of epochs can be easily found using training and validation loss as described in Section 2.3.2.

Apart from the mentioned Gradient Descent algorithms, there are many alternatives[1]. The most used methods are also gradient-based, but others such as Simulated Annealing [33] or Evolutionary Programming [15] are derivative-free. These algorithms have their specific hyper-parameters, but their description is out of the scope of this thesis.

2.3 Hyper-parameters of Neural Networks

As mentioned before, the hyper-parameters influence the structure or the learning process of the NN and need to be explicitly set before training the network. They have a significant influence on the accuracy of the NN, so it is beneficial to know how they influence the NN in order to optimise them. Even though not all NNs share the same structure or use the same learning algorithms, they frequently use the same common hyper-parameters that have a similar influence on the resulting behaviour of the NN.

Common hyper-parameters that define NN structure are number of hidden layers, number of neurons in the hidden layer, and activation function. These hyper-parameters are

[1] https://en.wikipedia.org/wiki/Outline_of_machine_learning#Machine_learning_algorithms

further described in Section 2.3.1. Learning of the NNs is influenced by hyper-parameters such as learning rate, dropout, momentum, number of epochs, batch size and weight decay, further described in Section 2.3.2. Though features described below have specific characteristics, their effect may differ in individual networks.

2.3.1 Structure related Hyper-parameters

The number of hidden layers is shown in Figure 2.2 and it determines the complexity of a problem that is the NN able to solve. Networks with [16]:

- **no hidden layers** – can solve only linearly separable problems
- **one hidden layer** – can solve almost any problem that contains a continuous mapping from one finite space to another
- **two hidden layers** – can be used to model data with discontinuities such as saw tooth wave pattern
- **more than two layers** – have no theoretical reason to be used but in practice can achieve better results

So while the higher number of layers can improve the accuracy of the NN, using too many hidden layers may lead to problems such as overfitting or vanishing gradient.

The number of neurons in hidden layer is a main measure in ability of NN to learn a particular function. Too few hidden neurons can lead to inability to learn the function (underfitting), too many hidden neurons can lead to overfitting and increase of time needed to train the NN [16]. While it is possible to have different number of units in each hidden layer, many networks use the same number for every hidden layer. There are many rule-of-thumb methods, such as [16]:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

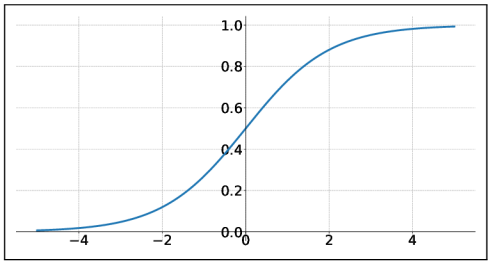
These suggestions can help with the selection of the number of hidden neurons, but they are more of a starting point than a rule.

Activation function determines the output of each layer in the network and has a major influence on network's accuracy, convergence and computational efficiency.

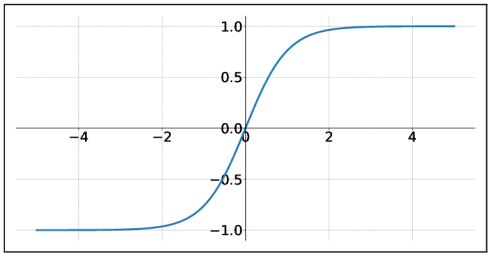
The activation function can be linear or non-linear. The linear activation functions can be represented by a straight line and have unconfined output. Because of the fact that linear combination of multiple functions is still a linear function, all subsequent layers with linear activation functions collapse into one. That is why the modern NNs use non-linear activations functions that enable the creation of deep NNs. It is common to use different activation functions for hidden layers and output layer, depending on what behaviour is desired. The hidden layers widely use ReLU activation functions for their features, which are described below. The activation function in the output layer depends on the desired output (i.e. whether it is for regression, classification, clustering, ...).

Different activation functions are used for their features which may be better in solving different problems. Sigmoid activation function has smooth gradient and provides clear predictions. For example as can be seen in Figure 2.4a, for x values outside of an interval $[-2, 2]$, y values are pretty close to 1 or 0. The output is confined on the interval between 0 and 1. Disadvantages of sigmoid function are that it's not zero centred, can lead to vanishing gradient problem (for very low/high x values) and is computationally expensive. In practice, sigmoid activation function is commonly used in output layer in classification problems.

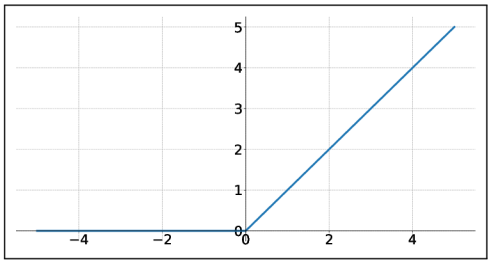
As can be seen in Figure 2.4b, hyperbolic tangent function has quite similar qualities as the sigmoid function. But it's zero centred, which means the output is in range $[-1, 1]$, strongly negative values are mapped to values close to -1, values close to 0 are mapped to values close to 0 and strongly positive values are mapped to values close to 1. That makes it easier to model inputs that have strongly negative, neutral and strongly positive values [24]. Hyperbolic tangent function is commonly used in classification problems.



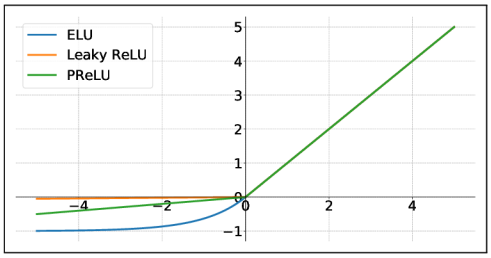
(a) Sigmoid



(b) Hyperbolic tangent



(c) ReLU



(d) Other ReLU-like functions

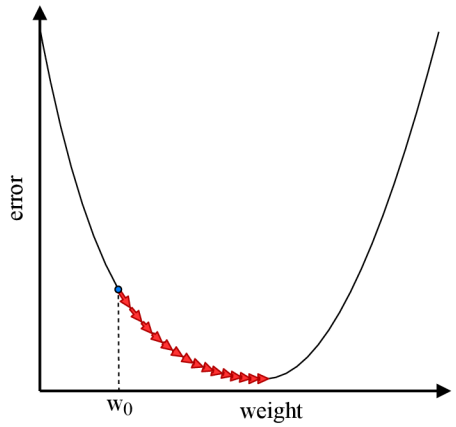
Figure 2.4: Common non-linear activation functions

Rectified Lineary Unit (ReLU) is a function where all negative values are mapped to zero and all positive values are mapped to identical values. ReLU is computationally efficient and the output is half-opened interval $[0, \infty)$. The problem of ReLU functions are zero and negative values, which are immediately mapped to zero (see Figure 2.4c). That decreases the ability of the model to fit or train from the data properly [27], which is known as the dying ReLU problem.

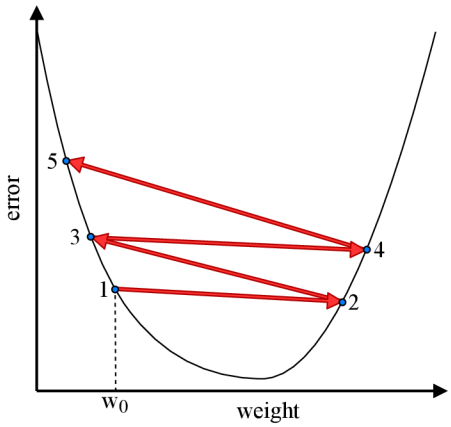
To prevent the dying ReLU problem there are multiple similar functions such as Leaky ReLU, Parametric ReLU [30] or ELU (Exponential Linear Unit) [11], which help increase the output range by having a small non-zero slope for negative values. Examples of such activation functions are in Figure 2.4d.

2.3.2 Learning related Hyper-parameters

Learning rate (LR) is used in gradient descent algorithm when parameters are updated according to an optimisation function [1]. Typically, values of the learning rate are a small positive numbers between 0 and 1. When optimising the learning rate, its values are usually sampled from log-space and suitable values are highly dependent on batch normalisation [5] that enables training with larger learning rate. Too low learning rate converges to the minimum smoothly, but slows down the learning process (Figure 2.5a). Too high learning rate speeds up the learning process, but may not converge (Figure 2.5b).



(a) Example of too low learning rate leading to slow convergence.



(b) Too high learning rate may cause oscillation around local minimum or even lead to divergence.

Figure 2.5: Problems of too low or too high learning rate. The red arrow represents single step in Gradient Descent algorithm, where each case used the same initial weight w_0.

Dropout is a regularization method that is used to prevent overfitting in NNs by randomly ignoring p neurons during the training phase. It means that for each training sample and each hidden layer, random fraction of hidden neurons are not considered. Then in testing phase, activation functions of the entire NN are considered, but each activation function is reduced by a factor p to account for the neurons ignored in training phase [36].

Weight decay is another regularization method that is used to prevent overfitting. It penalizes large weights by modifying the error function:

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|^2 , \tag{2.4}$$

where $\|\mathbf{w}\|^2 \equiv \mathbf{w}^T\mathbf{w} = w_0^2 + w_1^2 + \ldots + w_M^2$, and the coefficient λ governs the relative importance of the regularization term compared with the error term in error function $E(\cdot)$ [4]. Usual settings of coefficient λ range between logarithmic values of 0 and 0.1. When weight decay is too high, the model may never fit quite well. When weight decay is too low, it might not prevent overfitting [37].

Momentum controls how much the previous weight update influences the current weight update. This can speed up the learning process by making more significant update of weights when minimum is in the opposite direction of gradient. This might be especially helpful when optimisation reaches *plateau*, an area where the error function decreases very slowly and thus gradient is small. Also, momentum can help overcome local minimum as

shown in Figure 2.6. Momentum is a number between 0 and 1 and it is common to use values close to 1 (0.9, 0.99, etc.) [18]. Too small values have negligible effect and too big values are more likely to miss the optimum and lead to longer learning time.

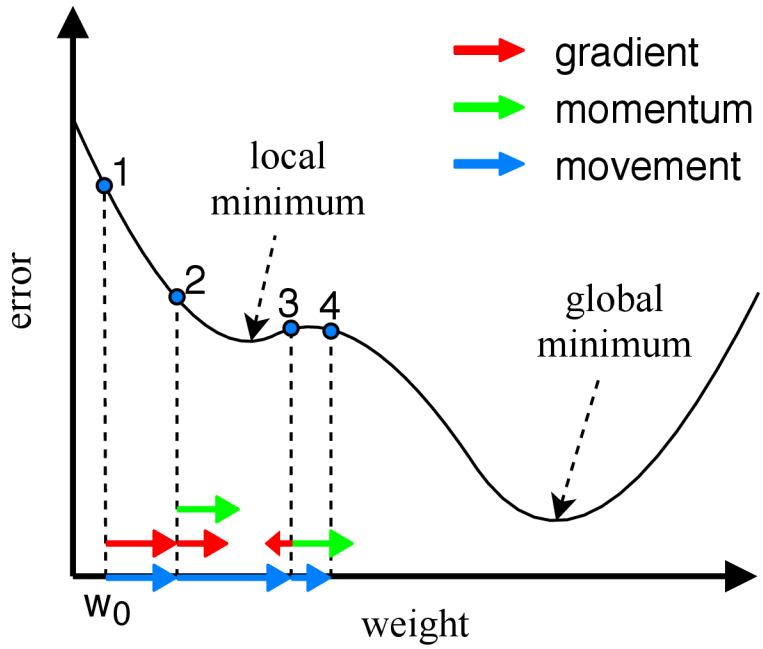


Figure 2.6: Example of how momentum can help Gradient Descent to get out of local minimum. Optimisation begins in point 1 with weight w_0. In this point momentum is 0 (since it is the first step) and the next weight update depends solely on the gradient and the learning rate. In the next steps, weight update is given by gradient and momentum addition. If momentum was lower that gradient in step 3, weight would shift back to the local minimum. But if momentum is significant enough, it can help get to other minima (though getting out of local minimum or getting to global minimum is not guaranteed).

Number of epochs is a number of complete cycles of learning algorithm through the whole dataset, which means it determines number of times the weights are updated. Too few epochs may cause underfitting, too many may lead to overfitting. Suitable number of epochs is usually found using training and validation loss, as shown in Figure 2.7. The goal is to find the highest possible number of epochs, before validation loss starts to grow because of overfitting.

Batch size defines number of samples used at once while training the NN. The range of batch size is from 1 up to a size of the training set. Smaller batch size causes more frequent model updates and allows more robust convergence that can lead to better accuracy, but can also lead to less accurate estimate of gradient (especially in more complex datasets) [18].

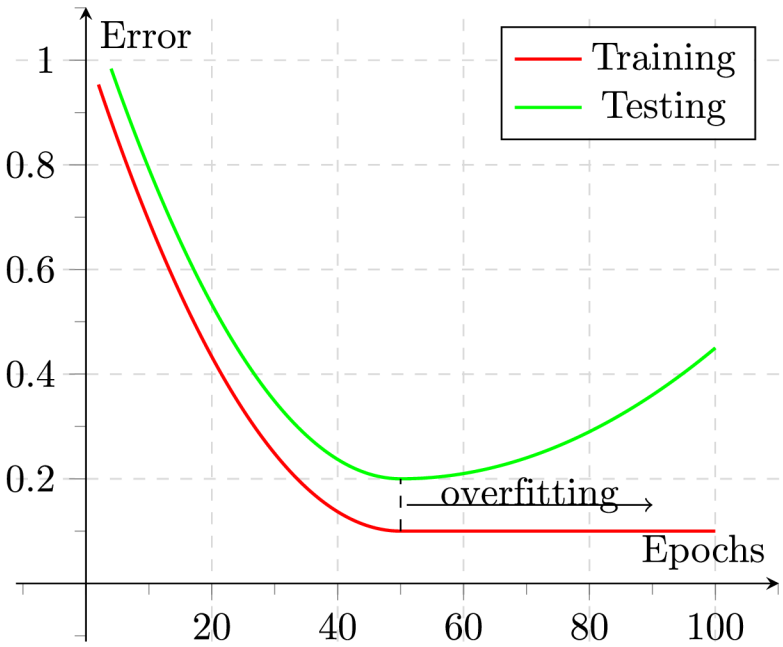


Figure 2.7: An illustrative example showing the influence of the number of epochs on the training and validation error (Adapted from Wikimedia Commons). Too many epochs can lead to overfitting, but it's possible to set suitable number of epochs based on the training and testing error. Suitable number of epochs is the highest possible number before the testing error starts to grow.

Chapter 3

Hyper-parameter Optimisation

Accuracy of a neural network is considerably influenced by configuration of its hyper-parameters. In order to improve the accuracy of a neural network, it is important to find acceptable values for these hyper-parameters. But since the time of training of a neural network might be quite substantial, hyper-parameters may be interdependent, and can acquire values from an infinite set, hyper-optimisation requires more complex solution than trying to find suitable values manually.

This chapter describes the essentials of neural network hyper-parameter optimisation. First section covers the problem of finding the optimal hyper-parameters of a neural network and describes different approaches of finding such hyper-parameter values. Next section covers basic principles of Gaussian processes. Section 3.3 describes how to solve regression problems using the Gaussian process model. Section 3.4 includes various approaches that can be used to estimate more accurate solution of the regression problem for given input.

3.1 Methods of Hyper-parameters Tuning

There are multiple common approaches in finding suitable hyper-parameter values. From the most straightforward as manual hyper-parameter tuning, Grid Search and Random Search optimisation [3] to more complex as Evolutionary optimisation or Sequential-Model Bayesian optimisation [13]. But since each neural network represents a specific problem, different approaches might fit some neural networks better than others and therefore it is beneficial for the user to learn about the features of each approach as well as specifics of optimised neural network to select suitable optimisation technique.

Also, it is important to take into account the amount of time the network needs to be trained. In case training the network doesn't take substantially more time then time needed to select hyper-parameter values for next optimisation step, it might be more efficient to use faster optimisation approach such as Random Search and make more optimisation steps instead.

The hyper-parameter optimisation problem can be defined as follows [21]:

Given a machine learning algorithm A having hyper-parameters $\mathbf{x} = x_1, ..., x_n$ with respective domains $\Lambda_1, ..., \Lambda_n$, we define its hyper-parameter space as $\Lambda = \Lambda_1 \times \cdots \times \Lambda_n$. For each hyper-parameter setting $\mathbf{x} \in \Lambda$, we use $A_\mathbf{x}$ to denote the learning algorithm A using this setting. We further use $\mathcal{L}(A_x, \mathcal{D}_{train}, \mathcal{D}_{valid})$ to denote the validation loss (e.g., misclassification rate) that $A_\mathbf{x}$ achieves on data $\mathcal{D}_{valid}$ when trained on $\mathcal{D}_{train}$. The hyper-parameter optimisation problem is then to minimise the blackbox function:

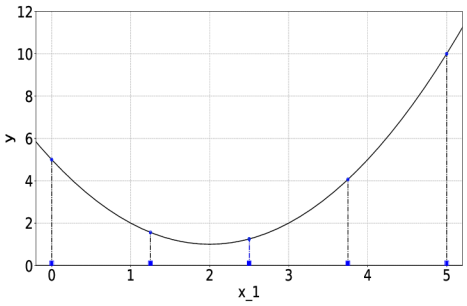
$$f(x) = \mathcal{L}(A_{\mathbf{x}}, \mathcal{D}_{train}, \mathcal{D}_{valid}) \tag{3.1}$$

The goal of the hyper-optimiser is to solve this problem in as few optimisation rounds as possible, where one optimisation round refers to invocation of the function $\mathcal{L}$ with hyper-parameter settings $\mathbf{x}$.

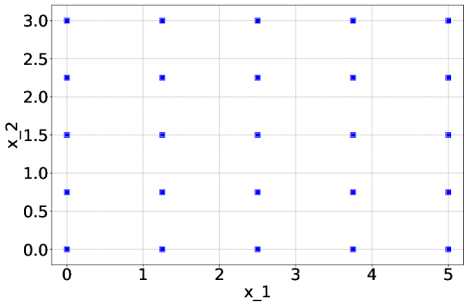
3.1.1 Grid Search Optimisation

Common solution in optimising the hyper-parameters is Grid Search. Based on number of hyper-parameters n and number of optimisation steps m that will be made, Grid Search optimisation algorithm selects a set X of m hyper-parameter settings $\mathbf{x_1}, ..., \mathbf{x_m}$. Each hyper-parameter setting $\mathbf{x}$ represents a point in n-dimensional space corresponding to a setting of each hyper-parameter. These points should be evenly spaced within predefined bounds so that visualization of selected points forms a grid, but the actual selection of the values may differ among various implementations of Grid Search algorithm. The result of the optimisation is the hyper-parameter setting $\mathbf{x}^* = \text{argmin} f(x), x \in X$.

For example, let's have hyper-parameter $\mathbf{h_1}$ with domain $[0, 5]$ and 5 optimisation steps. Suppose the influence of hyper-parameter $\mathbf{h_1}$ is given by function $y = (x-2)^2 + 1$, where y is network's loss. Then optimizer evenly splits the interval and selects 5 hyper-parameter settings $\mathbf{x_1}, ..., \mathbf{x_5}$ as shown in Figure 3.1a and trains the network for each selected setting of $\mathbf{h_1}$. The setting $\mathbf{x}^* = \mathbf{x_3} = (2.5)$ resulting in best NN accuracy is then the result of optimisation. Analogously for optimisation of two or more hyper-parameters. Assume additional hyper-parameter $\mathbf{h_2}$ with domain $[0, 3]$ and 25 optimisation steps. Then the optimiser uniformly selects training points as shown in Figure 3.1b and trains the network for each selected setting of the two hyper-parameters. Again, the setting with the best accuracy is the result.



(a) Optimisation of function $y = (x-2)^2 + 1$ using Grid Search.



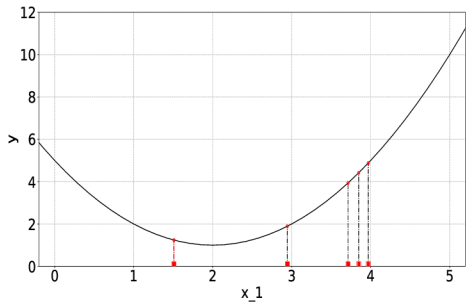
(b) Visualisation of search space while optimising two hyper-parameters using Grid Search.

Figure 3.1: Examples of using Grid Search in optimisation of one or two hyper-parameters.

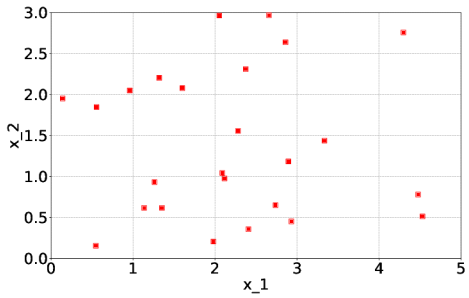
3.1.2 Random Search Optimisation

Random Search is an elementary optimisation technique, and remarkably, it was empirically and theoretically proved to be more efficient way of hyper-parameter optimisation than Grid Search [3]. Analogously as in Grid search, Random Search optimisation algorithm selects m settings $\mathbf{x_1}, ..., \mathbf{x_m}$, but the settings are selected randomly with uniform distribution across

Λ. An example of settings selection in optimisation of one or two hyper-parameters is shown in Figure 3.2.



(a) Optimisation of function $y = (x-2)^2 + 1$ using Random Search.



(b) Visualisation of search space while optimising two hyper-parameters using Random Search.

Figure 3.2: Examples of using Random Search in optimisation of one or two hyper-parameters.

3.1.3 Bayesian Optimisation Methods

Bayesian optimization is a set of powerful methods for optimizing objective functions which are very costly or slow to evaluate [6]. These methods keep a record of past evaluations of the objective function and create a probabilistic model that helps predict the function value for parameters that have not been yet evaluated.

The optimisation process is defined more closely by Sequential Model-Based Optimisation (SMBO), which is a formalism for Bayesian optimisation. Sequential refers to running trials one after another, where in each trial new hyper-parameter setting is found using Bayesian reasoning and updating a probabilistic regression model $\mathcal{M}$ [22]. SMBO process can be defined by following algorithm [12]:

Algorithm 1: Sequential Model-Based Optimisation

Input: $f, \mathcal{X}, S, \mathcal{M}$

1 $\mathcal{D} \leftarrow initSamples(f, \mathcal{X})$
2 **for** $i \leftarrow |\mathcal{D}|$ ***to*** T **do**
3 $\quad p(y|\mathbf{x}, \mathcal{D}) \leftarrow fitModel(\mathcal{M}, \mathcal{D})$
4 $\quad \mathbf{x}_i \leftarrow \text{argmax}_{\mathbf{x}\in\mathcal{X}} S(\mathbf{x}, p(y|\mathbf{x}, \mathcal{D}))$
5 $\quad y_i \leftarrow f(\mathbf{x}_i)$
6 $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_i, y_i)\}$
7 **end**

Input of the SMBO algorithm is the objective function $f(\cdot)$, domain $\mathcal{X}$ of the function $f(\cdot)$, acquisition function $S(\cdot)$ and probabilistic regression model $\mathcal{M}$. On the first line, a historical set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_i, y_i)\}$ is initialised with a few samples from the objective function, which are selected from domain $\mathcal{X}$. Steps on the lines 3–6 are executed in a cycle until the limit of function evaluations T is not met, including the function evaluations in $initSamples(\cdot)$ function. First, the probabilistic model $p(y|\mathbf{x}, \mathcal{D})$ is created based on the regression model $\mathcal{M}$ and historical set $\mathcal{D}$ (line 3). Then, this model is used to select new sample $\mathbf{x}_i \in \mathcal{X}$ by maximising the acquisition function $S(\cdot)$ (line 4). The method of selection

of the samples $\mathbf{x} \in \mathcal{X}$ for the probabilistic model will be further refered to as Domain Search Strategy (DSS) and is described in more detail in Section 3.4.4. Acquisition function (AF) is a method that is used to find next sample $\mathbf{x}_i$ by predicting which of the selected samples $\mathbf{x}$ will bring the best acquisition. More details about common acquisition functions are in Section 3.4.5. And finally, objective function $f(\mathbf{x}_i)$ is evaluated (line 5) and results is added to the historical set $\mathcal{D}$.

Methods in Bayesian optimisation can be differentiated based on their probabilistic regression model $\mathcal{M}$ [12]. Three of the most common regression models are Gaussian Processes (GP), Random Forests and Tree-Parzen Estimators (TPE).

Gaussian processes have become standard surrogate for modelling objective functions in Bayesian optimisation [35] and are described in detail in Section 3.2.

Random forests regression [26] is a supervised learning algorithm that uses combination of multiple simpler models – regression trees. The approach in hyper-optimisation is to construct a set of regression trees B and assume a Gaussian $\mathcal{N}(y|\hat{\mu}, \hat{\sigma})$ that models the probabilistic distribution $p(y|\mathbf{x}, \mathcal{D})$, where the parameters $\hat{\mu}$ and $\hat{\sigma}$ are chosen as the empirical mean and variance of the regression values $r(\mathbf{x})$ in the set of regression trees B[12]:

$$\begin{aligned} \hat{\mu} &= \frac{1}{|B|} \sum_{r \in B} r(x) \\ \hat{\sigma}^2 &= \frac{1}{|B| - 1} \sum_{r \in B} (r(\mathbf{x}) - \hat{\mu})^2 \end{aligned} \tag{3.2}$$

TPE regression models deviate from standard SMBO algorithm, since they apply Bayes rule to the models $p(\mathbf{x}, \mathcal{D}|y)$ and $p(y)$, instead of directly using the probabilistic model $p(y|\mathbf{x}, \mathcal{D})$. Probabilistic model $p(\mathbf{x}, \mathcal{D}|y)$ can be replaced with two non-parametric distributions, represented by processes $l(\mathbf{x})$ and $g(\mathbf{x})$ [12]:

$$p(y|\mathbf{x}, \mathcal{D}) = \begin{cases} l(\mathbf{x}) & \text{if } y < y^* \\ g(\mathbf{x}) & \text{if } y \geq y^* \end{cases} \tag{3.3}$$

where y^* is predefined threshold. The result is that TPE creates two different distributions for the parameters, density $l(\mathbf{x})$ formed by using the observations $\mathbf{x}_i$ such that the corresponding loss y_i is less than the threshold, and density $g(\mathbf{x})$ when y_i is greater than the threshold [2].

3.1.4 More Hyper-optimisation Approaches

Apart from the before mentioned baseline solutions, there are other commonly used and computationally effective solutions. A lot of them are modifications or combinations of Random and Grid search, such as Random Walk or Random Grid search. Others, such as Greedy search, optimise hyper-parameters one-by one using some kind of heuristics.

More complex (and less computationally effective) approaches used to tackle the problem of hyper-parameter optimisation, such as Genetic Algorithms, Particle Swarm Optimisation or Simulated Annealing, are well known optimisation methods and are common alternatives to Bayesian optimisation.

3.1.5 Comparison of Hyper-optimisation Methods

It's not a trivial task to compare various hyper-optimisation methods. The efficiency of each algorithm differs based on the optimised algorithm, type of the solved problem (classification, regression, etc.), specifics of used dataset, provided information about optimised problem, default algorithm settings etc. Another important thing to mention is that a many of available experiments use different optimisers to compare the methods, therefore even two GP based optimisers with the same optimiser parameters might perform differently. The following paragraphs try to capture the trends seen in multiple studies and generalize them in order to compare general differences between methods described in previous sections.

When considering before mentioned straight-forward methods based on Random search, the results of individual methods are quite similar. But for example in a study [28] focused on hyper-optimisation of Recommender Systems, simple Random search and Random Grid search provided better results than Random Walk or Greedy search on all tested datasets. Also, these solutions are usually less computationally demanding and highly parallelizable, in contrast to more complex methods. Therefore, it might be beneficial to use these methods when evaluating less costly functions and make more optimisation steps.

In some specific cases, even Random search can outperform more complex algorithms. But more often, solutions such as Bayesian optimisation achieve better results in hyper-optimisation of different machine learning algorithms [34]. When comparing various Bayesian and other mentioned more complex methods it's hard to make any generalisations, because different methods have proven more efficient in different experiments [13][29].

One of the key differences between GP and other mentioned hyper-optimisation methods is in configurability. GP based methods require more knowledge to be configured (i.e. to find suitable kernel), but provide more control over the created model. In case the general behaviour of the solved problem is known and the GP parameters are set to fit this model, it could theoretically provide better results and therefore might be beneficial to use. Naturally if the GP parameters are set poorly, the model provides worse results.

Since hyper-optimisation is specific to the optimisation task, no hyper-optimisation method has proven to be most efficient in general and it's up to the user to select suitable optimisation method.

3.1.6 Existing hyper-optimisation tools

Since each neural network represents a specific problem, different approaches might fit some neural networks better than others. Therefore, many of available optimisers provide a wide rage of hyper-optimisation techniques from simplest solutions as grid search or random search to a complex parallel computing solutions based on machine learning approaches.

There are many solutions that aim to solve hyper-parameter optimisation problem[1]. Some of the solutions are specific to certain language or library (e.g. *talos*[2] or *SHERPA*[3] for Keras), some provide more general solutions enabling to optimise almost any defined problem (e.g. *Hyperopt*[4]). Naturally, optimisers specific to a library enable to optimise

[1] https://medium.com/@mikkokotila/a-comprehensive-list-of-hyperparameter-optimization-tuning-solutions-88e067f19d9

[2] https://github.com/autonomio/talos

[3] https://github.com/sherpa-ai/sherpa

[4] https://github.com/hyperopt/hyperopt

only neural networks written in such library, but are usually easy to use and might have more control over optimised network.

Another distinction between available optimisers is whether are they designed to run on a single machine or rely on cloud computing resources. Optimisers that are heavily parallelized and use multiple machines are usually not free to use and are generally harder to set-up, but provide substantially more computing power and therefore resources to realize more optimisation rounds in the same time.

High-level solution is provided by Google's internal tool *Vizier* [17], a scalable black-box optimisation engine with remote procedure call interface, wide selection of optimisation algorithms and dashboard. *Advisor*[5] is an open-source implementation of Google Vizier and offers easy to use API with JSON configuration files to define specifics about NN training and hyper-parameter optimisation. It supports running trials on distributed systems, it's not library dependent and provides around 15 optimisation methods, including Grid search, Random search, Bayesian optimisation, Simulated Annealing and others.

Other similar, high-level tools are Microsoft's Neural Network Intelligence (NNI), HiPlot from Facebook and Ray Tune. All of the mentioned tools provide an API for visualisation of hyper-optimisation results, support running trials on distributed systems and some of the tools are even open-sourced.

Another widely used hyper-optimisation tool is Hyperopt, which is a Python library that supports parallelization using MongoDB or Apache Spark. The hyper-optimisation algorithms implemented in Hyperopt, such as Random Search or Simulate Annealing, are commonly used by other tools (Advisor, Hyperas).

3.2 Gaussian Process

Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a *stochastic process* describes the properties of functions [31]. A stochastic process $y(\mathbf{x})$ is specified by giving the joint probability distribution for any finite set of values $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_d)$ [4]. A Gaussian process is a stochastic process, where $p(y(\mathbf{x}_1), \ldots, y(\mathbf{x}_D)|\mathbf{x}_1, \ldots, \mathbf{x}_D)$ is D-dimensional Gaussian distribution.

3.2.1 Multivariate Gaussian distributions

The basic building block of a Gaussian process is the multivariate Gaussian distribution, where each random variable is distributed normally and their joint distribution is also Gaussian [19]. The multivariate Gaussian distribution is defined by mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. Mean vector $\boldsymbol{\mu}$ defines the centre around which distribution revolves, while covariance matrix $\boldsymbol{\Sigma}$ models variance along each dimension and defines correlation between every two random variables.

Assume a vector $\mathbf{X}$ of d random variables $x_1 \ldots x_d$ that follows normal distribution:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{3.4}$$

[5] https://github.com/tobegit3hub/advisor

Then mean $\boldsymbol{\mu}$ is d-dimensional vector given by expected value of each respective random variable x_i:

$$\boldsymbol{\mu} = \mathrm{E}[\mathbf{X}] = (\mathrm{E}[x_1], \mathrm{E}[x_2], \ldots, \mathrm{E}[x_d]) \tag{3.5}$$

and covariance matrix $\boldsymbol{\Sigma}$ is $d \times d$ matrix defined as:

$$\boldsymbol{\Sigma} = \mathrm{cov}[\mathbf{X}] = \mathrm{E}[(\mathbf{X} - \mathrm{E}[\mathbf{X}])(\mathbf{X} - \mathrm{E}[\mathbf{X}])^{\mathrm{T}}] \tag{3.6}$$

Note that the diagonal of covariance matrix $\boldsymbol{\Sigma}$ consists of the variance σ_i^2 of the i-th random variable x_i and the off-diagonal elements $\sigma_{ij}, i \neq j$ describe the correlation between random variables x_i and x_j [19]:

$$\Sigma_{ij} = \mathrm{cov}[x_i, x_j] = \mathrm{E}[(x_i - \mathrm{E}[x_i])(x_j - \mathrm{E}[x_j])^{\mathrm{T}}] \tag{3.7}$$

The covariance $\mathrm{cov}[x_i, x_j]$ of two random variables x_i and x_j is defined by a *kernel function*, which determines the characteristics of the resulting probability distribution [19]. Kernel function is probably the most important parameter of GP and it's described in more detail in Section 3.4.2.

3.2.2 Conditional And Marginal Gaussian Distributions

There are two important properties of the multivariate Gaussian distribution that are a key to Gaussian processes – *conditioning* and *marginalisation*. Both conditioning and marginalisation work with joint probability of two subsets of original random variables, which will be denoted as:

$$P_{X,Y} = \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_Y \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{XX} & \boldsymbol{\Sigma}_{XY} \\ \boldsymbol{\Sigma}_{YX} & \boldsymbol{\Sigma}_{YY} \end{bmatrix} \right) \tag{3.8}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are subsets of original random variables, mean $\boldsymbol{\mu}_X$ ($\boldsymbol{\mu}_Y$) corresponds to mean vector of subset $\mathbf{X}$ ($\mathbf{Y}$), matrix $\boldsymbol{\Sigma}_{XX}$ ($\boldsymbol{\Sigma}_{YY}$) corresponds to covariance matrix $\mathrm{cov}[\mathbf{X}]$ ($\mathrm{cov}[\mathbf{Y}]$) and matrixes $\boldsymbol{\Sigma}_{XY}$, $\boldsymbol{\Sigma}_{YX}$ correspond to covariance matrixes:

$$\begin{aligned} \mathrm{cov}[\mathbf{X}, \mathbf{Y}] &= \mathrm{E}[(\mathbf{X} - \mathrm{E}[\mathbf{X}])(\mathbf{Y} - \mathrm{E}[\mathbf{Y}])^{\mathrm{T}}] \\ \mathrm{cov}[\mathbf{Y}, \mathbf{X}] &= \mathrm{E}[(\mathbf{Y} - \mathrm{E}[\mathbf{Y}])(\mathbf{X} - \mathrm{E}[\mathbf{X}])^{\mathrm{T}}] \end{aligned} \tag{3.9}$$

Gaussian distribution is closed under conditioning, which means that if two sets of random variables are jointly Gaussian, then the conditional distribution of one set conditioned on the other is also Gaussian [4]. So if two subsets $\mathbf{X}$ and $\mathbf{Y}$ of original random variables follow normal distribution, then $\mathbf{X}|\mathbf{Y}$ and $\mathbf{Y}|\mathbf{X}$ are also Gaussian and are defined as [19]:

$$\begin{aligned} \mathbf{X}|\mathbf{Y} &\sim \mathcal{N}(\boldsymbol{\mu}_X + \boldsymbol{\Sigma}_{XY}\boldsymbol{\Sigma}_{YY}^{-1}(\mathbf{Y} - \boldsymbol{\mu}_Y), \boldsymbol{\Sigma}_{XX} - \boldsymbol{\Sigma}_{XY}\boldsymbol{\Sigma}_{YY}^{-1}\boldsymbol{\Sigma}_{YX}) \\ \mathbf{Y}|\mathbf{X} &\sim \mathcal{N}(\boldsymbol{\mu}_Y + \boldsymbol{\Sigma}_{YX}\boldsymbol{\Sigma}_{XX}^{-1}(\mathbf{X} - \boldsymbol{\mu}_X), \boldsymbol{\Sigma}_{YY} - \boldsymbol{\Sigma}_{YX}\boldsymbol{\Sigma}_{XX}^{-1}\boldsymbol{\Sigma}_{XY}) \end{aligned} \tag{3.10}$$

Gaussian distribution is also closed under marginalisation, so the marginal Gaussian distribution P_X and P_Y from joint distribution $P_{X,Y}$ is also Gaussian:

$$\begin{aligned} P_X &= \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_{XX}) \\ P_Y &= \mathcal{N}(\boldsymbol{\mu}_Y, \boldsymbol{\Sigma}_{YY}) \end{aligned} \tag{3.11}$$

3.3 Using Gaussian Processes for Regression

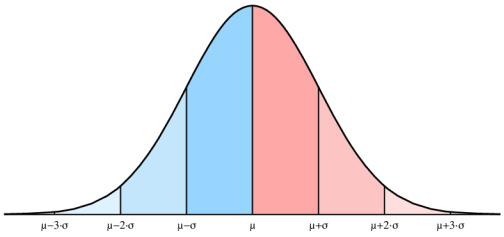
In order to solve regression tasks, GP needs to model an interpolation of observed data. This model is represented by joint probabilistic distribution $P_{X,Y}$ created from two discrete sets of random variables, where $\mathbf{Y} = (y_1, y_2, \ldots, y_n)^{\mathrm{T}}$ represents observed (training) data and $\mathbf{X} = (x_1, x_2, \ldots, x_m)^{\mathrm{T}}$ represents testing data. The testing data is used to sample a continuous function with a set of discrete points.

The joint probabilistic model $P_{X,Y}$ is a combination of the training and testing data, as shown in Equation 3.8. To make predictions about possible values of the testing points, conditional Gaussian distribution $P_{X|Y}$ is used. Thanks to the conditioning on the training data, resulting distribution limits the values of testing points that are close to any of the training points.

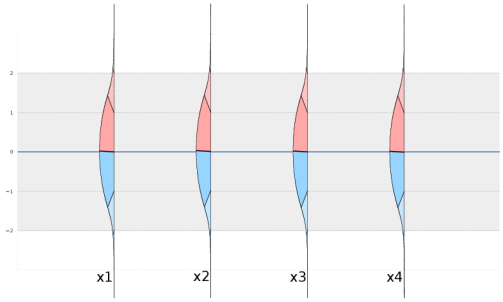
Note that in Gaussian processes, it's often assumed that $\boldsymbol{\mu}$ is a zero vector. This assumption simplifies the equations necessary for conditioning, while correction of mean can be done after making a prediction [19].

3.3.1 Prior distribution

In case no training data are available yet, distribution defined by GP will be $P_X = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\mu} = \mathbf{0}$ is a zero vector and $\boldsymbol{\Sigma}$ is covariance matrix with $m \times m$ dimensions. This is called a *prior distribution*. Its probability distribution could be visualised as in Figure 3.3, where each random variable x_i is normally distributed around 0. Functions sampled from this distribution would be Gaussian and their shape would be dependent on kernel function, as can be seen in Figure 3.8.



(a) Example of normal distribution. Normal distribution is centred around mean μ and can be separated into multiple sectors, based on the standard deviation σ.



(b) Probability distribution P_X, where each testing point is represented by a random variable x_i. Each random variable follows normal distribution and is centred around mean μ_i. Grey area represents space between $\mu_i - 2\sigma_i$ and $\mu_i + 2\sigma_i$ for each random variable x_i.

Figure 3.3: Visualisation of prior distribution over a set of testing points X.

3.3.2 Posterior distribution

Training data can be added to the GP model by forming a joint distribution $P_{X,Y}$, where $\mathbf{Y}$ is a set of random variables representing the training data. Then the posterior distribution $P_{X|Y} = (\boldsymbol{\mu}_{X|Y}, \boldsymbol{\Sigma}_{X|Y})$ defines distribution of testing data conditioned on training data. As can be seen in Equation 3.10, mean $\boldsymbol{\mu}_{X|Y}$ only depends on conditioned variable so the

posterior distribution is constrained to the set of functions that pass through the training points [19].

To illustrate, assume a prior distribution P_X in Figure 3.3b and training data $x_1 = 3.2, y_1 = 4.4$. Then the conditional distribution $P_X|y_1$ models the interpolation of the training data, as shown in Figure 3.4. Only one training value does not create a good interpolation of the data and therefore predicted values of the testing points that are further from the training data tend to return back to zero. This feature of GPs is further exploited in hyper-optimisation, because the best possible value of the loss function (error) is 0.

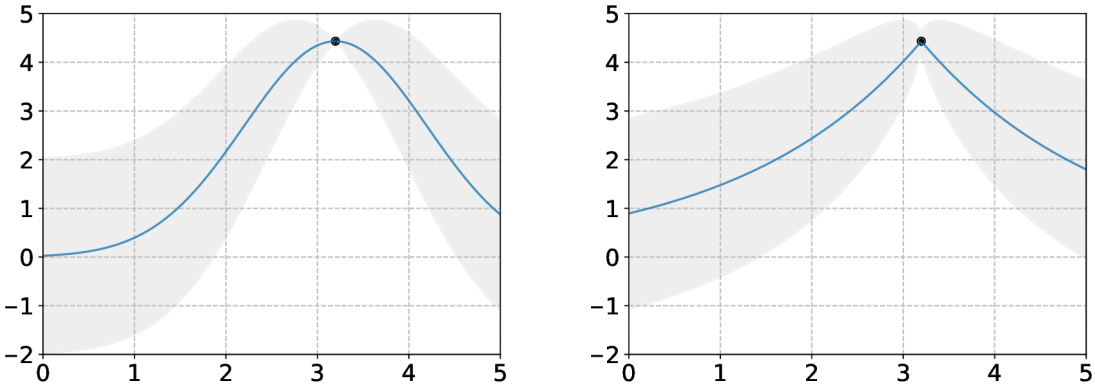


Figure 3.4: Examples of conditional probability distribution with one training data point $x_1 = 3.2$. Note that the shape and size of the distribution is defined by the covariance matrix.

3.3.3 Gaussian Processes in Hyper-optimisation

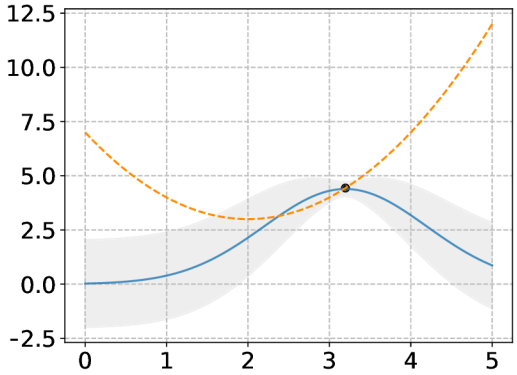
In order to understand how to use GPs in hyper-optimisation, let's connect the information from previous section to SMBO algorithm in Section 3.1.3. Assume objective function is a loss function $f(\cdot)$ and no initialisation samples are available, so the historical set $\mathcal{D}$ is empty. First, GP regression model $P_{X,Y}$ is created and DSS (see Section 3.4.4) is used to select the testing data (line 3). Note that according to the prior assumption, $Y = \emptyset$. Then, AF (see Section 3.4.5) creates conditional probabilistic model $P_{X|Y} = P_X$ (there are no training data) and uses this model to select next training point $\mathbf{y_1}$ (line 4). Loss function $f(\cdot)$ is evaluated in selected point $\mathbf{y_1}$ (line 5) and tuple $(\mathbf{y_1}, f(\mathbf{y_1}))$ is added to the historical set (line 6).

This is repeated until the limit of total function evaluations T is reached. Difference in following cycles is that the historical set $\mathcal{D}$ is not empty, so the shape of the distribution can be more helpful in selection of new training point $\mathbf{y_i}$.

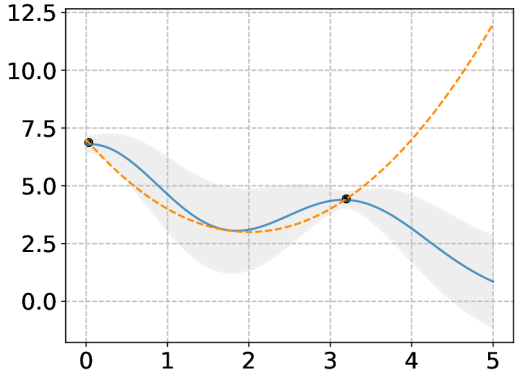
In terms of hyper-optimisation, each random variable x_i represents hyper-parameter setting of D hyper-parameters $h_1 ... h_D$. The training points are vectors of hyper-parameters values that have been already used to train the network. Therefore, the result of the loss function for those hyper-parameters settings are known. Testing points (in terms of hyper-optimisation) are settings of hyper-parameters, whose values is the optimiser trying to predict using the GP model. Training the NN represents the objective function, where training point is a parameter of the function and function value is resulting loss.

Figure 3.5 shows optimisation of one hyper-parameter on domain $[0, 5]$ with grid DSS and minimal mean acquisition function. First optimisation step selects new hyper-parameter

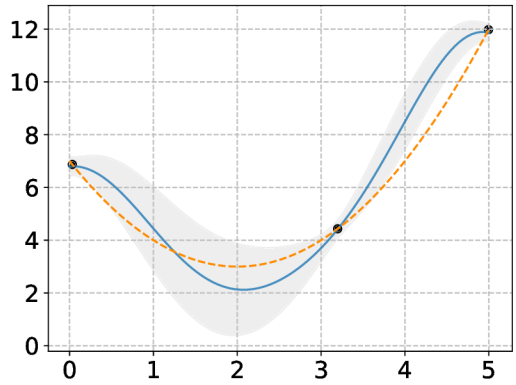
setting randomly, while next hyper-parameter settings are selected based on predicted value of mean.



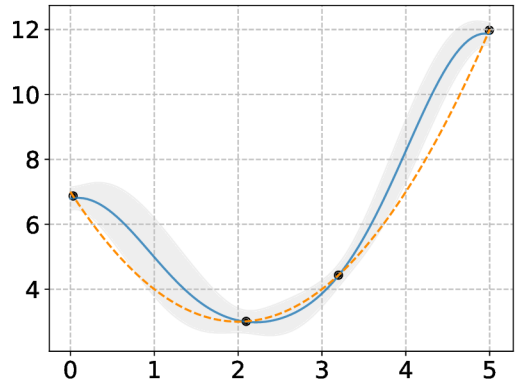
(a) Step 1: training point 3.2 selected randomly, next training point will be the one with minimal mean – 0



(b) Step 2: next training point will be 5



(c) Step 3: next training point will be 2.01



(d) Step 4: best found setting is 2.01

Figure 3.5: Four rounds of GP hyper-optimisation of one hyper-parameter, where dotted orange line represents real shape of the loss function.

Figure 3.6 shows optimisation of two hyper-parameters with the same settings as above. Value of each hyper-parameter is displayed on separate axis, while prediction is on axis z. Optimisation algorithm works the same, but kernel, DSS and AF have to be able to work with multidimensional points.

GPs can be used to optimise arbitrary number of hyper-parameters, but there are computational limitations. The computational complexity of GP regression is $\mathcal{O}(n^3)$, where n is the size of the training set. This is due to the inversion of the covariance matrix $\mathbf{\Sigma}_{YY}$ in computation of conditional probabilistic distribution $P_{X|Y}$. Moreover, optimising more hyper-parameters might lead to a bigger search space, so the model will need more training samples to create reasonable approximation and more testing samples to make more precise predictions.

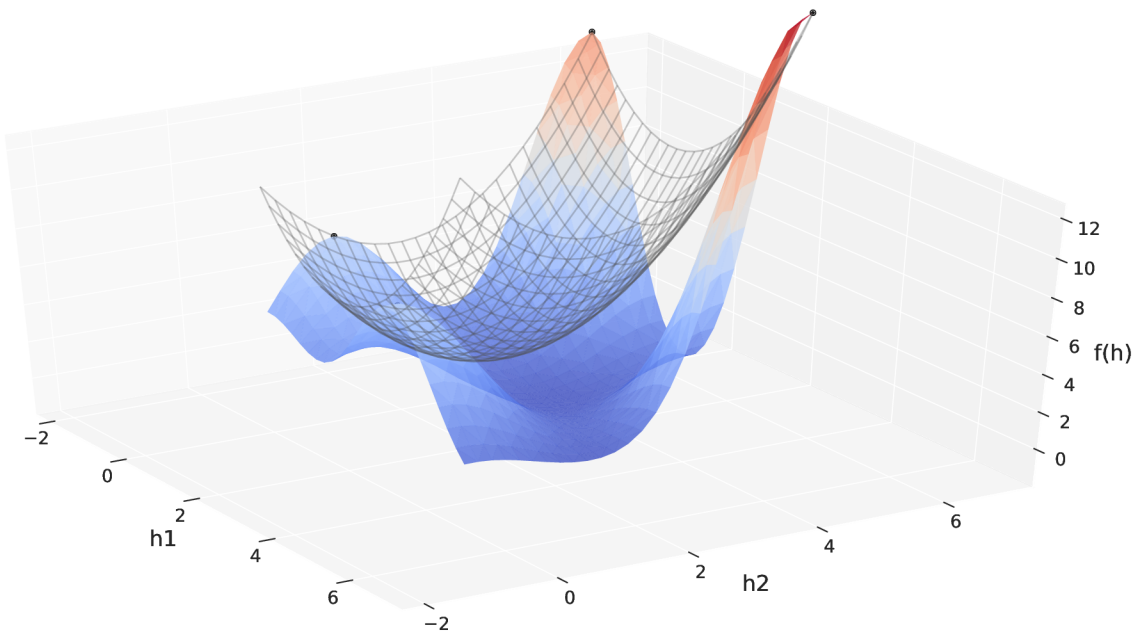


Figure 3.6: GP prediction after four rounds of optimisation of two hyper-parameters h_1 and h_2. Wireframe plot represents the real shape of approximated loss function $f(\cdot)$, while red and blue area represents predicted mean values given by conditional probabilistic distribution $P_{X|Y}$. Four training points $y_1 = (3.2, 0)$, $y_2 = (0, 5)$, $y_3 = (5, 5)$, $y_4 = (0, 1)$ are marked as a black dots.

3.4 Parametrization of GP optimiser

As well as other probabilistic regression models in Sequential Model-Based Optimisation (SMBO) described in Section 3.1.3, GP regression model has a number of parameters that can be changed in effort to improve the result of the optimisation. These parameters include selection of mean, uncertainty of measurement, kernel function and its parameters, Domain Search Strategy (DSS) and Acquisition Function (AF).

3.4.1 Mean and Uncertainty

As mentioned before, mean is usually set to zero. That means that in prior distribution, mean μ_i of all random variables x_i is zero. In posterior distribution, mean of all testing data will tend to return to zero more and more the further it is from any training data. In hyper-optimisation of NNs, zero is usually a good selection of mean. This is because the optimal function value of the loss function is zero and its desirable that mean returns to this value in space with no training points.

However, it might be desirable to use another mean value when minimising function with optimum that is lower or much higher than zero. Since assuming mean is a zero vector simplifies the computation of conditional probability distribution, it's practical to use zero mean for prediction and then shift the prediction to correct mean value [19].

It's also possible to edit the probability distribution in training points, so the potential functions do not have to intersect the exact location of the training point. This is done by

uncertainty parameter R, which is used to multiply the variance of each training point $\mathbf{y}_i$ in the covariance matrix:

$$\Sigma_{ii} = \text{cov}[\mathbf{y}_i, \mathbf{y}_i] \cdot R \tag{3.12}$$

and is subsequently added to covariance $\mathbf{\Sigma}_{XX}$ when calculating conditional probability distribution (Equation 3.10):

$$\mathbf{X}|\mathbf{Y} \sim \mathcal{N}(\boldsymbol{\mu}_X + \mathbf{\Sigma}_{XY}\mathbf{\Sigma}_{YY}^{-1}(\mathbf{Y} - \boldsymbol{\mu}_Y), \mathbf{\Sigma}_{XX} + R - \mathbf{\Sigma}_{XY}\mathbf{\Sigma}_{YY}^{-1}\mathbf{\Sigma}_{YX}) \tag{3.13}$$

The effect of uncertainty can be seen in Figure 3.7, where predicted mean no longer intersects some of the training points and the grey area given by standard deviation is wider. The bigger the uncertainty R is, the more significant inaccuracies are permitted by the probability distribution. The uncertainty parameter is particularly useful when there's a jitter in the data, so the probabilistic distribution is able to model these deviations.

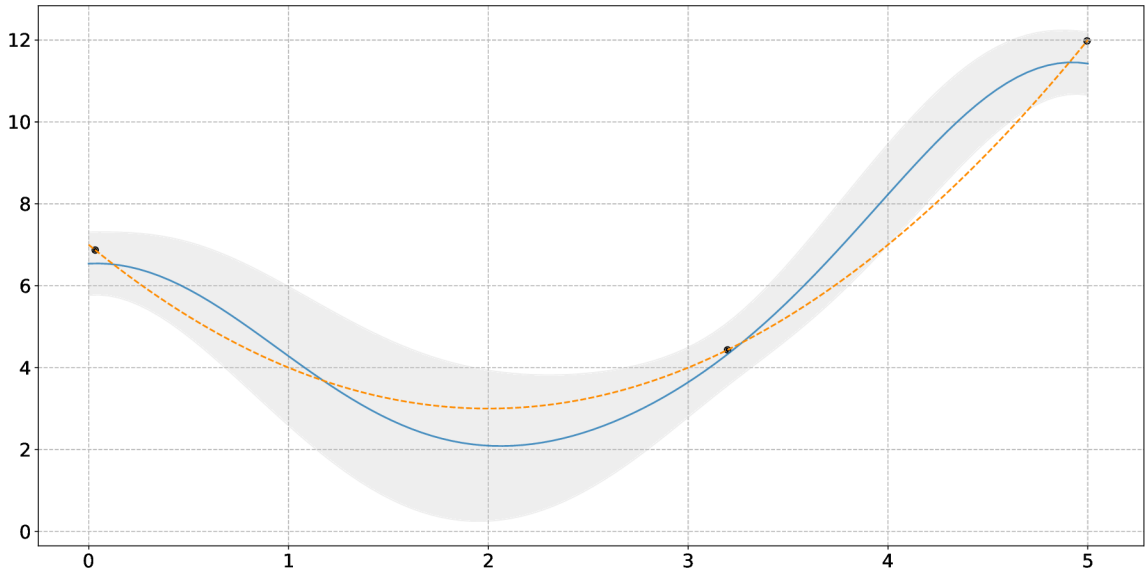


Figure 3.7: GP prediction (with uncertainty $R = 0.05$) after three rounds of optimisation of one hyper-parameter.

3.4.2 Kernels

The kernel (or kernel function) k is a measure that defines similarity between two D-dimensional points and is used to compute values Σ_{ij} in covariance matrix $\mathbf{\Sigma}$ [19]:

$$k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}, \Sigma_{ij} = k(\mathbf{x}, \mathbf{x}') \tag{3.14}$$

Selection of the kernel determines resulting shape of the probabilistic distribution (see Figure 3.4) as well as the behaviour of the functions sampled from this distribution (see Figure 3.8). Following paragraphs describe a few of the common kernel functions and their features.

Radial Basis Function (RBF) kernels, as the name suggests, are kernels that are based on radial basis functions. So, the result of such kernel function depends only on distance

between the two points. The RBF kernel uses Euclidean norm to measure the distance and is defined as follows:

$$k_{RBF}(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{1}{2}\frac{\sum_{i=1}^{D}(x_i - x_i')^2}{\sigma_l^2}\right), \tag{3.15}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are tested D-dimensional points and σ_l is *characteristic length scale* parameter. This parameter determines how distant the two points $\mathbf{x}$ and $\mathbf{x}'$ have to be for the function value to change significantly [31]. The second kernel parameter σ_f is *signal standard deviation*, which determines the uncertainty of Gaussian probability distribution in GP. Examples of functions sampled from GP with RBF kernel are shown in Figure 3.8a.

Equation of Laplacian kernel is very similar to RBF kernel, but it uses absolute value instead of Euclidean norm to determine the distance between the points $\mathbf{x}$ and $\mathbf{x}'$:

$$k_{Laplacian}(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{1}{2}\frac{\sum_{i=1}^{D}|x_i - x_i'|}{\sigma_l^2}\right) \tag{3.16}$$

Although as can be seen in Figure 3.8b, compared to RBF kernel, Laplacian kernel is more suitable to model functions with rapid local changes and sharp edges.

As can be seen in Figure 3.8, Matérn ARD kernel is by its behaviour very similar to RBF kernel. The reason is that Matérn kernel also commonly uses Euclidean norm to determine the distance of the points, but it's also altered by some additional constants and parameters:

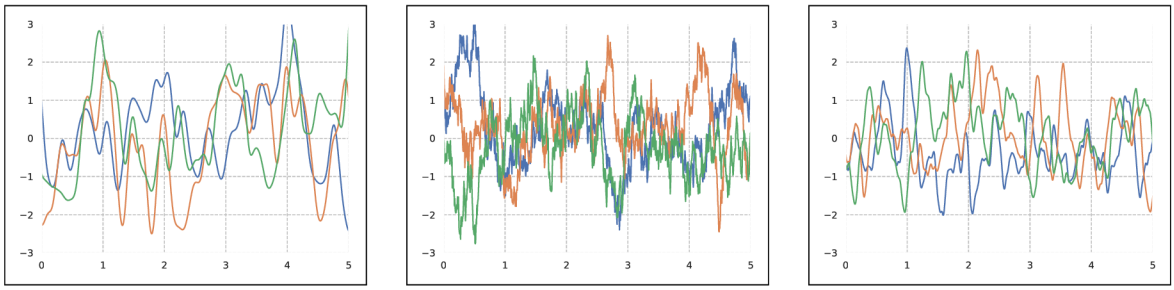
$$\begin{aligned} k_{Matern}(\mathbf{x}, \mathbf{x}') =& \sigma_f^2 \left(1 + \sqrt{5}r + \frac{5}{3}r^2\right) \exp\left(-\sqrt{5}r\right) \\ r =& \sqrt{\sum_{i=1}^{D} \frac{(x_i - x_i')^2}{\sigma_{li}^2}} \end{aligned} \tag{3.17}$$

The biggest advantage of this kernel in comparison with RBF kernel is its ability to reflect more substantial local changes. Also, it's a kernel with Automatic Relevance Determination (ARD), so it's able to define different characteristic length scale $\boldsymbol{\sigma}_l$ for each function parameter. This is particularly useful in cases when the same change in distinct hyper-parameters values leads to a different behaviour – i.e. one hyper-parameter changes the resulting accuracy of the network much more (or less) prominently than the other. ARD is widely used in many kernels and can be used in both RBF and Laplacian kernels.

Two of the most elementary kernels, constant and linear kernel, are not very suitable to be used separately, but they can be used to construct new kernels. Constant kernels $k_{const}(\mathbf{x}, \mathbf{x}') = c$ return predefined constant c independently on the input, while linear kernels multiply constant θ with the vector inputs:

$$k_{linear}(\mathbf{x}, \mathbf{x}') = \theta \mathbf{x}^T \mathbf{x}' \tag{3.18}$$

Even though above mentioned kernels are just a small fraction of the kernels that can be used, they might not fit some specific model. But there are many techniques for constructing new kernels. These techniques use existing kernel functions and modify them. Kernels created this way have distinctive features and are named after the modification that was performed. For example, additive kernels use addition to create new kernel k from two existing kernels k_1 and k_2:



(a) RBF (b) Laplacian (c) Matérn ARD

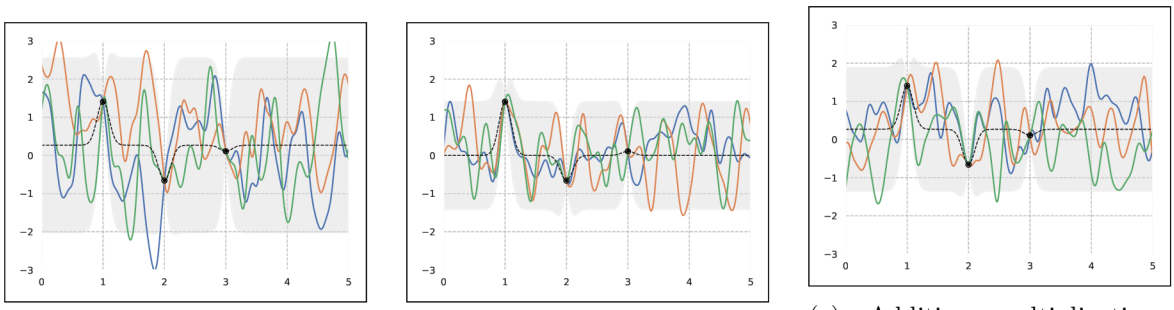
Figure 3.8: Examples of random function samples using different kernels.

$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}'), \tag{3.19}$$

while multiplicative kernels use kernel multiplication:

$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') \cdot k_2(\mathbf{x}, \mathbf{x}') \tag{3.20}$$

But many more building methods such as exponentiation and composition can be used to create new kernels. These methods can be used repeatedly or combined together to create more complex kernel that fits the solved problem better, as shown in Figure 3.9.



(a) Additive kernel (RBF + constant) (b) Multiplicative kernel (RBF · constant) (c) Additive multiplicative kernel ((RBF + constant) · constant)

Figure 3.9: Examples of kernels created by kernel addition, kernel multiplication or their combination. By adding RBF and constant kernel, it's possible to stretch the distribution and move mean to the average function value of training points, while preserving features of RGB kernel. Multiplicative kernel can change the standard deviation so the Gaussian is wider or narrower. And by combination of the kernel building methods, it's possible to combine features of all kernels.

3.4.3 Automatic Tuning of Optimisation Parameters

The selection of parameters of the optimiser such as kernel and uncertainty might be a difficult task, but it is possible to automatically tune those parameters. The automatic tuning is build on evaluation of the log likelihood function $p(\mathbf{y}|\boldsymbol{\theta})$, where $\mathbf{y}$ is a vector of function values of the training points and $\boldsymbol{\theta}$ is a set of tuned parameters [4]. The log

likelihood function for a GP model is evaluated using the standard form for a multivariate Gaussian distribution, given by equation [4]:

$$\ln p(\mathbf{y}|\boldsymbol{\theta}) = -\frac{1}{2}\ln|\boldsymbol{\Sigma}| - \frac{1}{2}\mathbf{y}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{y} - \frac{N}{2}\ln(2\pi) \tag{3.21}$$

Log likelihood of the multivariate Gaussian distribution is then maximised using some optimisation algorithm. In each round of the optimisation algorithm, values in covariance matrix $\boldsymbol{\Sigma}$ are recalculated using updated parameters $\boldsymbol{\theta}$ and used to calculate log likelihood from Equation 3.21 above. This optimisation is called Maximum Likelihood Estimation and can be used to optimise all numeric parameters of the optimiser.

3.4.4 Domain Search Strategies

Domain search strategy (DSS) is a method that is used by optimiser to select m testing samples $\mathbf{X} = (\mathbf{x}_0, \ldots, \mathbf{x}_m)$. These testing samples are then used to create GP with joint probabilistic distribution $P_{X,Y}$. Two most common methods are based on grid search and random search.

Grid search creates a D-dimensional grid of k points, where $k = m_{dim}^D \wedge k \geq m$ and m_{dim} is the number of samples per dimension. Subsequently, m samples from the grid are selected. Random search selects samples randomly with uniform distribution. The strategy as well as the number of selected samples has significant influence on resulting probabilistic distribution.

3.4.5 Acquisition functions

As stated before, an Acquisition function (AF) is a function that is used to find next training point $\mathbf{y}_i$ from a set of testing points $\mathbf{X}$. AF in GPs uses predictions from conditional probability $P_{X|Y}$ and selects the point with best acquisition. This acquisition is usually based on exploration-exploitation trade-off. The result of the AF is a point with the best acquisition and is used for training the NN.

The most straightforward approach is strictly improvement based, which means that the acquisition depends only on predicted value. It selects the point $\mathbf{x}^*$ based on predicted *minimal mean* value:

$$\mathbf{x}^* = \operatorname{argmin}\, \mu(\mathbf{X}) \tag{3.22}$$

This is a special case of confidence bound AFs, such as Sequential Design Optimisation function, where the acquisition is given by *Lower Confidence Bound:*

$$LCB(\mathbf{x}) = \mu(\mathbf{x}) - \kappa\sigma(\mathbf{x}), \tag{3.23}$$

and the confidence parameter κ is set to zero.

Another acquisition function, *Probability of Improvement*, introduces the exploration-exploitation trade-off parameter ξ and is defined as:

$$PI(\mathbf{x}) = \Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})}\right), \tag{3.24}$$

where $\mathbf{x}^+$ is so far the best discovered point with value $f(\mathbf{x}^+)$ and Φ is Cumulative Distribution Function (CDF) [7]. The selected point would then be the one with the lowest acquisition $PI(\cdot)$, since the goal is to minimise the function:

$$\mathbf{x}^* = \text{argmin } PI(\mathbf{X}) \tag{3.25}$$

The most used acquisition function is *Expected Improvement* (EI), which selects value with the best expected improvement given by equation:

$$\begin{aligned} \text{EI}(\mathbf{x}) &= \begin{cases} (\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi)\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & \text{if } \sigma(\mathbf{x}) > 0 \\ 0 & \text{if } \sigma(\mathbf{x}) = 0 \end{cases} \\ \text{where } Z &= \begin{cases} \frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})} & \text{if } \sigma(\mathbf{x}) > 0 \\ 0 & \text{if } \sigma(\mathbf{x}) = 0 \end{cases} \end{aligned} \tag{3.26}$$

where ϕ is Probability Density Function (PDF) [7]. The selected point is again the one with the lowest acquisition $EI(\cdot)$:

$$\mathbf{x}^* = \text{argmin } EI(\mathbf{X}) \tag{3.27}$$

The selection of suitable AF depends on the specific model that is used and problem that is being solved, therefore it's not an elementary task to do. But there are strategies such as portfolio allocation that use multiple AFs and have proven to be almost always more effective than individual AFs [8].

Chapter 4

Design and Implementation of Toolkit for Hyper-optimisation of Neural Networks

This chapter describes design, implementation, usage and possible customisation of Gaussian Process based Optimiser (GPOP), a toolkit for hyper-optimisation of neural networks.

As stated before, commonly available tools for hyper-optimisation can be separated into two categories: library specific and universal. Even though library specific approach offers more control over the optimised NN, I have chosen to design universal hyper-optimiser. Apart from the obvious advantage of being able to optimise almost any NN, this approach does not require any user's knowledge about the programming language or libraries used to implement the NN and thus makes the optimiser easier to use.

4.1 Toolkit Design

Apart from optimising multiple possibly correlated hyper-parameters, the goal was to design a toolkit that is lightweight and easy to use. GPOP aims to provide the latter through a library, that enables control over the optimisation process and creates an interface between the optimiser and the NN. Moreover, it's possible to run the optimisation using a *CLI wrapper* that is controlled through a set of command line arguments making it possible to run optimisation without writing any code. The usage and structure of the toolkit are shown in Figure 4.1 and described in more detail in following paragraphs.

The centrepiece of the toolkit is the *optimiser*. Optimiser has no knowledge about what is optimised and views every optimised problem as a black-box function with a certain number of parameters specified in the beginning of the optimisation. Optimiser interacts with the *function interface* in order to find the best parameters of the black-box function and therefore best hyper-parameter settings. The function interface serves only as an interface for optimised problem that ensures every optimiser's call to evaluate a black-box function is uniform.

A counterpart component interacting with the function interface is the *bridge*. The purpose of the function interface and the bridge is to create an interface between the NN and the optimiser, so that the optimiser can treat the NN as if it was a function and receive results of the loss function as a single value. So when the bridge receives parameters from the optimiser, it creates a new subprocess that trains the NN with hyper-parameters

corresponding to those parameters. After the subprocess ends, the bridge collects the output of the training, finds the result (loss) and returns it to the optimiser as a function call result.

Running the subprocess and collecting the result is possible thanks to a user defined *configuration file* that contains information about how to run the training of the network and how to get the result. Moreover, the configuration file contains more information about optimised hyper-parameters, such as domain ranges and numeric types.

For the purpose of experiments with behaviour of various optimisation techniques, there is *benchmarks* component. This component contains a set of benchmark functions that simulate some of the typical problems that hyper-optimisation incorporates. And since the actual shapes of the functions are known, they can be useful to learn about the features of the used optimisation technique before optimising hyper-parameters of the actual NN.

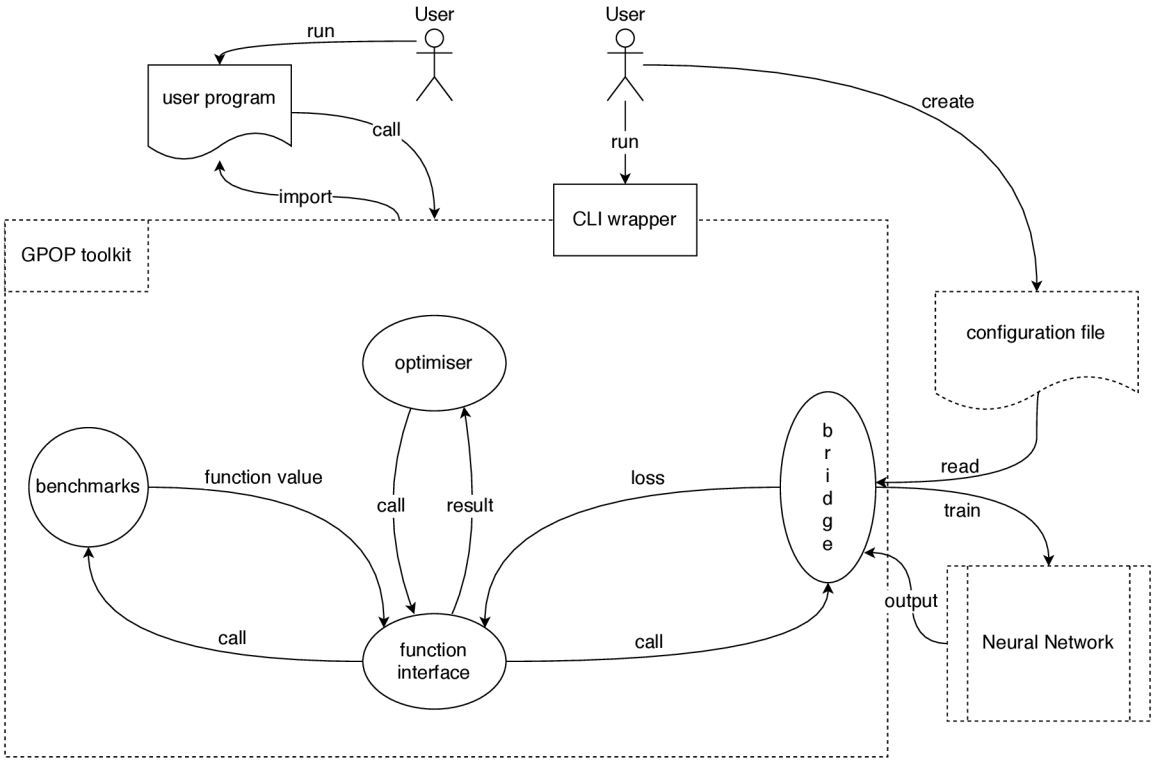


Figure 4.1: Usage of the toolkit: GPOP provides user with two different methods of usage – run optimiser from command line with CLI wrapper or import the library in a custom user program. Running from command line does not require any custom code except configuration file, but doesn't provide any extra information about optimisation. Library use, on the other hand, enables user more control over the optimisation, including the access to the GP model and creation of custom kernels in order to learn and adjust the optimiser to specific problem.

4.2 Toolkit Implementation

The toolkit is implemented as a collection of Python3 modules that provide classes and methods for hyper-optimisation of NNs. Moreover, it contains a python wrapper script enabling configurable optimisation. The toolkit is split in three Python packages:

optimiser is core package containing all modules needed during the optimisation itself, including kernels, acquisition functions, etc.

nnbridge is a package providing resources to create an interface between the optimiser and the NN, enabling the optimiser to train the NN and get result of the training. Furthermore, it contains methods for simple manipulation with NN configuration files.

tests package contains unit tests for each module of the GPOP toolkit.

Relations between individual modules and packages is shown in Figure 4.2, while detailed description of all modules can be found in sections below.

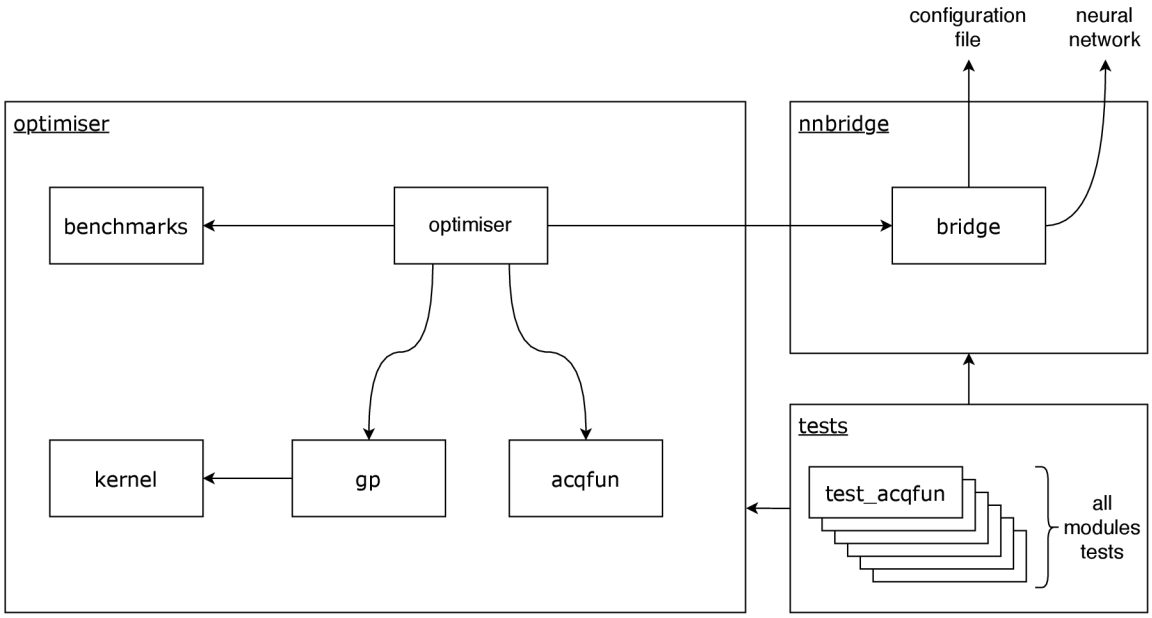


Figure 4.2: Distribution of modules into packages and relations between individual modules and other components. For example module `optimiser` uses most of the implemented modules, except `kernel` and test modules. Module `bridge` doesn't use any modules, but other components like configuration file and neural network.

4.2.1 Gaussian Process Representation

Module `gp` contains implementation of Gaussian process. They are represented by `GP` class, which creates a model of a Gaussian process given by its covariance matrix $\mathbf{\Sigma}$ and training points $\mathbf{Y}$ (mean $\boldsymbol{\mu}$ is assumed to be a zero vector). Covariance matrix is calculated as follows:

```
cov_matrix = np.ones(N, N)
for i in range(N):
    for j in range(i, N):
        cov = k(Y[i], Y[j])
        cov_matrix[i][j] = cov
        cov_matrix[j][i] = cov
cov_matrix = cov_matrix + R * np.eye(N)
```

where N is the number training points $\mathbf{Y}$, k selected kernel and R is uncertainty of measurement:

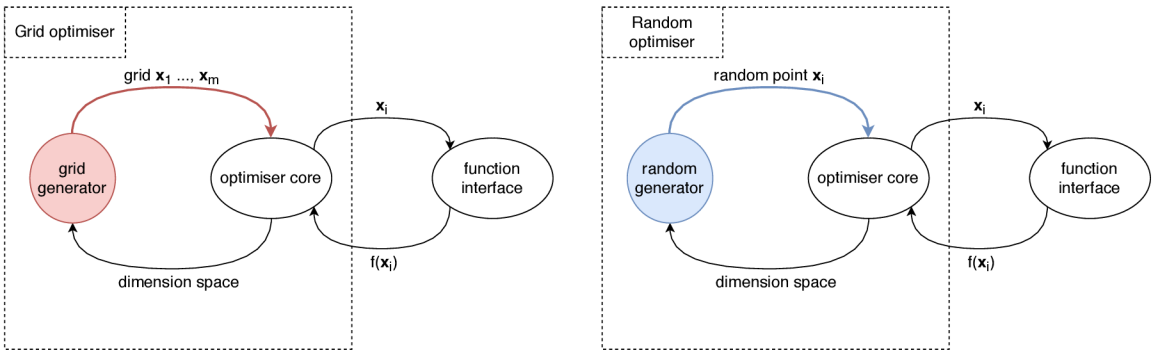
The key method is `predict(x)`, which returns conditional multivariate Gaussian probability distribution for the point `x` represented by its mean value `mean_x` and standard deviation `std_dev_x`. This probability distribution is computed using conditioning from the covariance matrix `cov_matrix` above:

```
def predict(x):
    cov = 1 + R * k(x, x)
    sigma_YX = np.zeros(N, 1)
    for i in range(N):
        sigma_YX[i] = k(Y[i], x)
    mean_x = (sigma_YX.T * cov_matrix.I) * Y_val.T
    std_dev_x = cov + R - (sigma_YX.T * cov_matrix.I) * sigma_YX
    return mean_x, std_dev_x
```

Thanks to this prediction, AF is able to determine next training point. For the detailed description of covariance matrix computation and predictions of testing points values, see Sections 3.2 and 3.3.

4.2.2 Optimisers

Module `optimiser` implements GP optimiser and two baseline optimisers that are based on random search and grid search. Even though the baseline optimisers were primarily meant to serve as a comparison to GP based optimisation, they can in some cases provide better results and therefore were included in the toolkit. The implementation of both baseline optimisers is quite straightforward, as shown in Figure 4.3.



(a) Grid (b) Random

Figure 4.3: Structural diagram describing essential parts of each optimiser.

The main component of both baseline optimisers is *optimiser core*. The goal of the core is to select as many training points as defined by user. Grid optimiser uses *grid generator* that creates a grid of m D-dimensional points using Cartesian product of D linearly spaced vectors, where D is dimensionality of the solved problem. Since the number of the training points n selected by user can be lower than the number of the points in the grid, the core selects first n points and passes them successively to the function interface. The point with the lowest function value $f(\mathbf{x_i})$ is returned as the result of the grid hyper-optimisation. Since the grid was formed using Cartesian product, computational complexity of the Grid optimiser is polynomial with respect to the number of training points.

The Random optimiser uses *random generator* instead of *grid generator*. The core of the Random optimiser uses this generator to select as many points as defined by user. *Random generator* generates one D-dimensional training point within *dimension space* with uniform probability distribution. Generated training point $\mathbf{x}_i$ is then passed to the function interface. This random point generation is repeated n times to generate enough training points. Finally, the point $\mathbf{x}_i$ with the lowest function value $f(\mathbf{x_i})$ is returned as the result of the hyper-optimisation. The computational complexity of random generator is linear, with respect to the number of training points.

The required arguments for both Grid optimiser and Random optimiser initialisation are *function* and *dimensions bounds*. The function is either benchmark or `NeuralNet` object with one parameter. This parameter is a 1D array representing settings of the optimised hyper-parameters. Dimensions bounds is a 2D list, where each inner list contains two numbers. They represent minimal and maximal values that hyper-parameter can acquire. Both baseline optimisers also have a keyword argument `verbose`, which if set to `True`, will ensure printing of additional hyper-optimisation information to the standard output.

The most complex optimiser is GP optimiser. The parameters of the black-box function that will be chosen for the next optimisation round depend on selected domain-space search strategy (DSS), acquisition function (AF) and predictions received from created `GP` model. As shown in Figure 4.4, the *core* is responsible for control of other components to find the best possible parameters of the black-box function. The result of the optimisation is again the training point with the lowest loss value, but GP optimiser object stores other data that are important for hyper-optimisation analysis and visualisation, including current GP model, used training points and their values. The computational complexity of GP optimiser is cubic, with respect to the number of training points.

Apart from a reference to the implemented *kernel* object, GP optimiser has the same required arguments as baseline optimisers. To specify additional parameters of GP optimiser such as DSS or AF, any of the keyword arguments described in Table 4.1 can be used.

4.2.3 Kernels

Kernels are implemented in separate module, along with builder methods allowing user to construct new kernels. It is possible to add new custom kernel, build new kernel from existing ones or use one of the five implemented kernels: RBF, Laplacian, Matérn ARD 5/2, constant and linear. All implemented kernels are described in detail in Section 3.4.2.

The parameters of the kernels can be automatically tuned during the optimisation, as described in Section 3.4.3. And in order to enable automatic parameter tuning, kernels work with tensors from `torch.Tensor`. Each kernel is implemented as callable class, where method `__call__` accepts two tensors as parameters representing two D-dimensional

Table 4.1: Keyword arguments of GP optimiser.

Parameter	Description
`x_init`	1D numpy array representing initial hyper-parameter configuration. If not set, random configuration will be used.
`R`	Uncertainty of GP model. If not set, default value 1e−5 will be used.
`nb_samples`	Number of testing sample per dimension (total number of samples: $nb\_samples^D$). If not set, default value 50 will be used.
`dss`	Used Domain Search Strategy. If not set, default strategy `random` will be used.
`acq_fun`	Used Acquisition Function. If not set, default function `ExpectedImprovement` will be used.
`htypes`	A list of hyper-parameter types. Permitted values are `„float"` for real numbers or `„int"` for integers. If not set, all hyper-parameters will be treated as real numbers.
`autotune`	If set to `True`, automatic tuning of kernel parameters is activated. Default value is `False`.
`autotune_all`	If set to `True`, automatic tuning of all parameters is activated. Default value is `False`.
`verbose`	If set to `True`, additional hyper-optimisation information is printed to standard output. Default values is `False`.

points. All parameters of the kernels are stored during initialisation to a class variable called `params`.

RBF and Laplacian kernels are implemented based on Equations 3.15 and 3.16 and have only one parameter – characteristic length scale σ_l. Signal standard deviation is fixed to value 1.0 and none of the kernels support ARD.

Kernel Matérn ARD 5/2 is implemented based on Equation 3.17 and has $D + 1$ parameters, first is signal standard deviation and the rest corresponds to characteristic length scale (one for each dimension D).

Last two implemented kernels are linear and constant kernel and each has only one parameter. Linear kernel is initialised with parameter θ, which is multiplied with kernel inputs. Constant kernel is initialised with constant value c, which is also its return value for every input.

Any of the kernels can be initialised directly using its class name or use function `selectKernel(name, params)`, where `name` is a string corresponding to a class name of the kernel and `params` is a tensor containing the parameters of the kernel.

GPOP also provides two methods to build new kernels – `buildAddKernel` and `buildMultKernel`. The first one creates a new kernel by addition of two existing kernels, while the second method uses multiplication.

In case user requires different kernel, it is possible to extend GPOP by implementing custom kernel or adding a kernel builder, as described in Section 4.4.3.

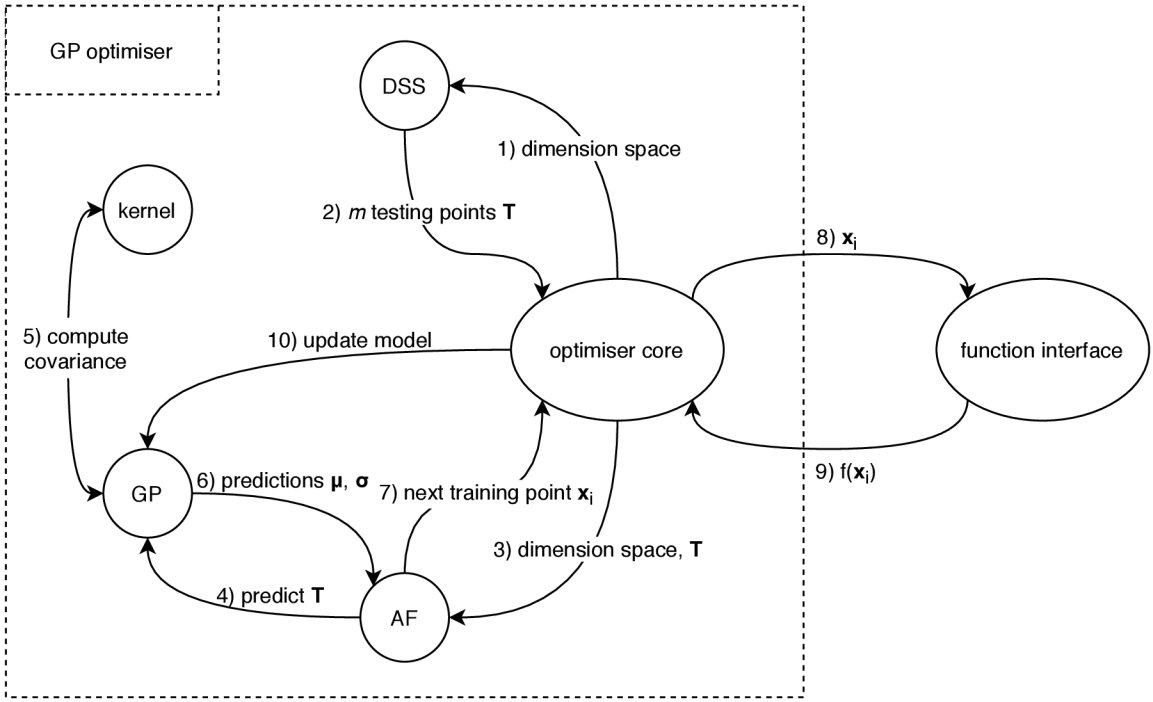


Figure 4.4: Structural diagram describing the function of GP optimiser: First, the core uses DSS to select a set of m testing points $\mathbf{T}$ within *dimension space*. Number of testing points depends on number of dimensions (hyper-parameters) to be optimised and number of samples per dimension defined. Next, acquisition function is used to select next training point $\mathbf{x}_i$. To make this selection, all acquisition functions use GP predictions in testing points $\mathbf{T}$. Finally, training point $\mathbf{x}_i$ is passed to *function interface*, the function value $f(\mathbf{x_i})$ is collected and GP model is updated. This whole process is repeated as many times as is the number of optimisation rounds.

4.2.4 Autotune

The automatic tuning of parameters of the optimiser is implemented as described in Section 3.4.3 and uses *Pytorch* framework with Adam optimiser to maximise the log likelihood of the GP model.

Automatic parameter tuning can be enabled by using command line options `--autotune` or `--autotune-all` in CLI wraper or by enabling one of the keyword arguments `autotune` or `autotune_all` in initialisation of GPOptimiser. Automatic tuning starts after at least 5 training samples are present and is repeated for 50 rounds or until the difference of log likelihood in two subsequent rounds is not lesser then predefined constant ϵ.

4.2.5 Domain-space Search Strategies

GPOP toolkit provides user with two distinct DSSs to select testing points: `grid` and `random`. These strategies are implemented inside `optimiser` module in the same manner as grid and random generators described in Section 4.2.2, but generated points are used as testing samples.

DSS can be selected by a keyword argument `dom_search_strategy` in initialisation of `GPOptimiser` or using a command line option `--domain-search-strategy` when using the CLI wrapper described in Section 4.2.8.

4.2.6 Acquisition Functions

Module `acqfun` implements three AFs: `MinimalMean`, `LowerConfidence` and `ExpectedImprovement`. Each AF is implemented as callable class with four positional parameters. First parameter is a list of testing points from DSS, second is GP model and the last two are lists of training points and their function values. All AFs use those parameters to predict mean values $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ of the testing points and select the next training point.

`MinimalMean` selects the point $\mathbf{x}^*$ with the lowest predicted mean value. The second method, `LowerConfidence`, selects the next training point $\mathbf{x}^*$ with minimal predicted lower confidence bound corresponding to:

$$\mathbf{x}^* = \text{argmin}(\mu(\mathbf{x}) - 2\sigma(\mathbf{x})) \tag{4.1}$$

The last method, `ExpectedImprovement`, selects the point with the best expected improvement defined in Equation 3.26.

Any of the AFs can be initialised directly using its class name or use function the `selectAcqFun(name)`, where `name` is a string corresponding to a class name of the AF. All implemented AFs are described in more detail in Section 3.4.5.

4.2.7 Bridge

Module `bridge` creates an interface between the optimiser and the NN and also provides user with methods to parse configuration files. First, the `bridge` creates callable `NeuralNet` objects, which behave like a standard functions, but when called, they actually run a subprocess in order to train the NN using specific hyper-parameter settings. After the subprocess ends, the `bridge` finds the result of the training in the output of the subprocess and returns it to the optimiser. The information needed to do so is stored in configuration file with following syntax:

```
SYNTAX:                                  EXAMPLE:
<command>                                python3 ./mnist/main.py --epochs 5
<result_regex>                           Average loss: ([-+]?\d*\.\d+|\d+)
<failure_regex>                          Average loss: (nan)
<HP-1_switch> <type> <from> <to>         --lr float 0 1
<HP-2_switch> <type> <from> <to>         --nb-hidden int 2 512
...
<HP-n_switch> <type> <from> <to>
```

First line is a command that will be used to run training of the NN and it can contain additional hyper-parameters that won't be optimised. Second line contains a regular expression that identifies desired result of the training in the output of the previously run command. This regular expression needs to specify the number that will be minimised by capturing it in a group. If more lines of the output match the regular expression, the bridge uses the last matched number. In case this regular expression does not match any text of

the output, regular expression `failure_regex` is used to identify whether training was run successfully. If this regular expression matches any text of the output, predefined default value is returned to the optimiser. Otherwise, an error indicating unsuccessful invocation of the subprocess is thrown. This safety check is useful when training is not properly invoked and therefore it is not necessary to continue hyper-optimisation. Example of such behaviour might be bad configuration of a hyper-parameter in configuration file, such as use of invalid value (e.g. negative learning rate). Then the training fails, the optimisation is stopped and the user is notified about the error.

The rest of the lines in the configuration file is reserved for the hyper-parameters that will be optimised, where each line defines one hyper-parameter and consists of four fields separated by space. First field, `HP-i_switch`, is string value used as a switch to identify i-th hyper-parameter inside the command. Second field, `type`, determines which values are assigned to this hyper-parameter by optimiser. Supported types are `int` for integer values and `float` for real numbers. Last two fields are reserved for specification of hyper-parameter's domain range.

This configuration is used to initialise the optimiser and then to run the subprocess using a command in form:

```
SYNTAX:
    <command> <HP-1_switch> <HP-1_val> ... <HP-n_switch> <HP-n_val>
EXAMPLE:
    python3 ./mnist/main.py --epochs 10 --lr 0.5 --nb-hidden 256
```

where `command` is the command from the configuration file and fields `<HP-1_val>`, ..., `<HP-n_val>` are values of the respective hyper-parameters selected by optimiser.

Configuration files can be parsed using one of the two ways: either from the standard output or from a file.

4.2.8 Command Line Wrapper

The Command Line Interface (CLI) wrapper `wrapper.py` is a Python3 script that is able to load and parse the configuration files, create and run the optimiser, plot cumulative minimum and Gaussian process representation and print hyper-optimisation information to standard output. To specify what tasks should be executed, CLI wrapper has a number of command line options that are described in detail in Table 4.2.

The command line option `--gaussian-process` has zero or two and more arguments. If no argument is specified, RBF kernel with default parameters and default uncertainty is used. The first argument is a class name of the selected kernel, the second argument is the uncertainty and the rest of the arguments correspond to kernel parameters in order defined by individual kernels (see Section 4.2.3).

Command line option `--benchmark` (or `-b`) has $D+3$ arguments. The first argument is a string representing the name of the benchmark. The second argument is a string containing an optimum $\mathbf{x}_{opt} = (x_1, \ldots, x_D)$, where each element x_i is separated by a space as shown in usage example in Section 4.3.1. The third argument is the function value in optimum $f(\mathbf{x}_{opt})$. Next D arguments define dimensions bounds for each dimension of the benchmark, where the minimum is separated from the maximum by a colon.

Note that options `-b` and `-f` can't be used together and option `-s` has no effect without the option `--draw`. For usage examples of the CLI wrapper, see Section 4.3.1.

Table 4.2: Command line options of the CLI wrapper script `wrapper.py`.

Long option	Short option	Description
--help	-h	Shows help message with usage information and exits.
--rounds	-r	Number of hyper-optimisation rounds. If not set, default value 10 will be used.
--grid-search	none	Use Grid search optimiser.
--random-search	none	Use Random search optimiser.
--gaussian-process	none	Use GP optimiser.
--domain-search-strategy	none	Selection of DSS (options: grid, random).
--acquisition-function	none	AF selection (options: MinimalMean, LowerConfidence, ExpectedImprovement).
--autotune	none	Enable automatic tuning of kernel parameters.
--autotune-all	none	Enable automatic tuning of all GP parameters (kernel + uncertainty).
--benchmark	-b	Optimise selected benchmark function.
--config-file	-f	Optimise NN defined by configuration.
--jitter	none	Add jitter given by option's argument to the benchmark function.
--draw	-d	Draw cumulative minimum and GP model plots.
--verbose	-v	Print additional hyper-optimisation information to standard output.
--round-by-round	none	Run hyper-optimisation one round at a time (draw plots and print output after every round).
--save-fig	-s	Save plots of cumulative minimum and GP model representation to a directory specified by options argument.
--seed	none	Set seed for randomised elements in the optimiser.

4.3 Usage Examples

As stated before, GPOP toolkit can be used either through CLI wrapper, or as a Python library. This section contains usage examples of GPOP toolkit with implemented benchmark functions and example MNIST NN [1]. The complete set of examples is stored in GPOP git repository[2] in folder `examples`.

To run the examples, install GPOP toolkit using `setup.py` script located in the root folder or run the examples from `gpop` subdirectory.

4.3.1 Command Line Interface

To run hyper-optimisation from the command line, use Python3 script `wrapper.py`. The hyper-optimisation task, used optimiser and further options can be selected using script's command line options. Full description of the wrapper script and it's command line options is in Section 4.2.8.

[1] https://github.com/pytorch/examples/tree/master/mnist
[2] https://gitlab.com/mcoufal/gpop

To run benchmark hyper-optimisation, command line option `--benchmark` has to be specified. The example below runs 9 optimisation rounds of Random optimiser on Ellipsoidal benchmark. This benchmark will be two-dimensional, with optimum $\mathbf{x}_{opt} = (5, 5)$ and optimal function value $f(\mathbf{x}_{opt}) = 3$. Both parameters will be optimised within the interval $[0, 10]$. Also, additional hyper-optimisation information, such as selected training points in each optimisation round, will be printed to standard output.

```
./wrapper.py \
--rounds 9 \
--random-search \
--benchmark "FnEllipsoidal" "5 5" 3 "0:10" "0:10" \
--verbose
```

To run NN hyper-optimisation, command line option `--config-file` has to be specified. This command line option has one optional argument, which provides a path (relative or absolute) to the configuration file of the NN. If this command line option is used without any argument, the configuration is taken from the standard input.

The example below runs 5 rounds of hyper-optimisation on specified NN using GP optimiser. The NN and optimised parameters are defined in file `mnist_example_config.txt`. The example of such configuration file is in Section 4.2.7. GP optimiser will use RBF kernel with characteristic length scale $\sigma_l = 0.1$, 15 testing samples per dimension and uncertainty $R = 0.01$. After the hyper-optimisation ends, plots of GP representation and cumulative minimum over rounds will be shown.

```
./wrapper.py \
--rounds 5 \
--gaussian-process "KernelRBF" 15 0.01 0.1 \
--config-file mnist_example_config.txt \
--draw
```

4.3.2 Usage of GPOP Python Library

The following examples demonstrate how to use GPOP as a library. First, the following imports need to be made:

```
import torch
from gpop.optimiser import optimiser as gpo
from gpop.optimiser import kernel
```

In case only Grid optimiser or Random optimiser will be used, only `optimiser` module needs to be imported. GP optimiser requires `kernel` module to create a kernel and `torch` module to define the parameters of the kernel.

If the hyper-optimisation task is to optimise a benchmark, `benchmarks` module has to be imported and benchmark function created:

```
from gpop.optimiser import benchmarks

# benchmark settings
bench_name = "FnSphere"
bench_opt = [2.0, 5.0]
bench_opt_val = 2
# create benchmark function
fn = benchmarks.selectBenchmark(bench_name, bench_opt, bench_opt_val)
```

In case the task is to optimise hyper-parameters of the NN, `bridge` module has to be imported and `NeuralNet` object initialised using a parsing function from the `bridge` module:

```
from gpop.nnbridge import bridge

# NN settings
config_path = "./mnist_example_config.txt"
fn, dim_bounds = bridge.parseFileConfig(config_path)
```

The configuration can be loaded from standard input using `parseStdInConfig()` function or from a file using `parseFileConfig()` function as above. These functions return a reference to a callable `NeuralNet` object and a list of dimensions bounds.

Next, the optimiser has to be initialised. Using its keyword arguments, it's possible to specify additional settings such as DSS or AF. Full description of available keyword arguments for each optimiser can be found in Section 4.2.2. Example below uses GP optimiser with Matérn ARD 5/2 kernel. The parameters of the kernel are signal standard deviation $\sigma_f = 1.0$ and characteristic length scale $\boldsymbol{\sigma}_l = (1.2, 1.5)$. Note that this kernel expects two-dimensional points as input:

```
kernel_name = "KernelMaternARD52"
params = torch.tensor([1.0, 1.2, 1.5], requires_grad=True)
k~= kernel.selectKernel(kernel_name, params)
opt = gpo.GPOptimiser(fn, dim_bounds, k, autotune=True, verbose=True)
```

Note that if the parameters of the kernel are supposed to be automatically optimised, the keyword argument `requires_grad` in kernel parameters initialisation needs to be set to `True`. In case a benchmark is optimised, the dimensions bounds need to be set manually:

```
dim_bounds = [[0.0, 5.0], [0.0, 5.0]]
```

Finally, the optimisation can be run for n rounds and the results of the optimisation can be shown:

```
# run optimisation
opt.optimise(n)
# print results
print("Best settings: hp1:", opt.best_x[0], "hp2:", opt.best_x[1])
print("Best function value:", opt.best_y)
```

Optimiser stores the best found settings $\mathbf{x}^*$ along with best found function value $f(\mathbf{x}^*)$ in variables `best_x` and `best_y`. The optimiser also stores other information, such as every used training point, its function value and values of automatically tuned parameters.

4.4 GPOP Customisation

In order to adjust GP optimiser to specific NN, it's possible to create custom kernels and acquisition functions. Also, it is possible to create new benchmarks that may help to select suitable hyper-optimisation parameters that suite specific NN.

To add a new kernel, AF or a benchmark, it's possible to edit existing Python modules `kernel`, `acqfun` and `benchmarks`, or to create a new module. But in order to preserve full functionality, it's recommended to use the existing modules.

4.4.1 Writing new kernels

In order to add a new kernel, create a new class in module `kernel` module. Also, it's important to maintain following rules to ensure GPOP works as expected:

- kernel has to be a callable class with initialisation parameter `params` stored as a class variable
- `params` is a tensor, where each element or group of elements represents a parameter of the kernel
- inputs of the kernel are two tensors, representing two D-dimensional points
- all operations on inputs and parameters have to use torch framework
- the new kernel has to be added to `selectKernel()` function

It's also possible to create new kernel builders, which should return a new kernel that follows the same conventions as mentioned above. For examples of kernels builders, see `buildAddKernel()` and `buildMultKernel()` functions in `kernel` module.

4.4.2 Adding New Acquisition Functions

To create a new AF, add a new class to `aqfun` module that follows rules below:

- AF has to be a callable class
- AF has three required input arguments:
  - `x_test` – an array of D-dimensional testing points
  - `x` – a list of D-dimensional training points
  - `y` – a list of function values for each training point in list `x`
- the outputs of AF are:
  - `x_next` – an array representing a D-dimensional point that will be used as next training point
  - `mean` – a list of predicted mean values of the testing points
  - `sigma` – a list of predicted standard deviations of the testing points
- the new AF has to be added to `selectAcqFun()` function

Note that the names of the inputs and outputs can be changed, but positions and types have to be preserved.

4.4.3 Adding New Benchmarks

To add a new benchmark, create a new class in module `benchmarks`. New benchmarks have to inherit from the `Benchmark` class and follow the rules below:

- benchmark has to be a callable class
- benchmark has two required positional initialisation arguments:
  - `x_opt` – a list of D values, representing an optimum of the benchmark
  - `f_opt` – a function value in the optimum of the benchmark
- the input of the benchmark is D-dimensional array, representing a point in space
- the output of the benchmark is a scalar
- the new benchmark has to be added to the function `selectBenchmark()`

Chapter 5

Experiments

The actual optimal values of hyper-parameter settings in neural networks are not known and moreover, the training time of the NN might be substantial. Therefore, it is common to test the performance of hyper-optimisation on benchmarks first [14]. Benchmarks are functions that are designed to simulate the typical difficulties that can occur during hyper-optimisation while searching the hyper-parameters domains [20]. To evaluate the efficiency of the implemented optimizers and their features, experiments were first conducted on benchmarks and then on a NN using MNIST dataset [25].

This chapter contains description of implemented benchmarks and the results of the conducted experiments.

5.1 Benchmarks for Hyper-parameter Optimisation

Since function parameters can be viewed as a particular hyper-parameters with continuous domains and the resulting function value as a result of the loss function, the benchmark functions are applicable substitute for actual NN. The advantage is that the minimal input vectors of such functions are known and can be even directly specified in the benchmark. Two basic benchmarks, Sphere and Ellipsoidal [20], were used in the experiments. Both represent unimodal, separable problems and are scalable with dimension. Unimodal means that the function has only one local minimum (maximum) and it's the global minimum (maximum). The separability of *D-dimensional* problem means that it can be separated into D one-dimensional procedures and solved independently.

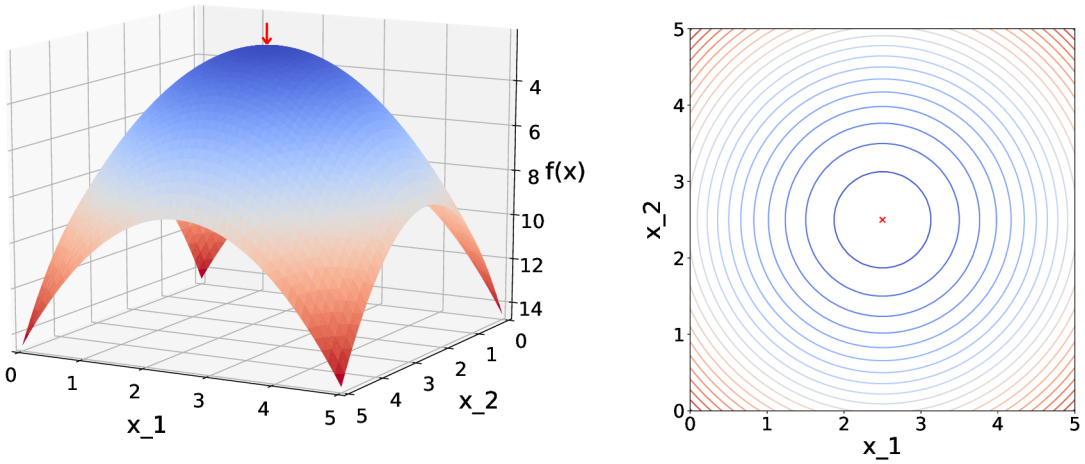
Defined benchmarks use the following notation: $\mathbf{x}$ is *D-dimensional* input vector, $\mathbf{x}^{opt}$ is an optimal solution vector and f_{opt} is minimal function value such that $f_{opt} = f(\mathbf{x}^{opt})$.

5.1.1 Sphere Function

Sphere function represents easy continuous domain search problem that is unimodal and highly symmetric. Sphere function is given as:

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{z}\|^2 + f_{opt} \\ \mathbf{z} &= \mathbf{x} - \mathbf{x}^{opt} \end{aligned} \tag{5.1}$$

An example of two-dimensional Sphere function is shown in Figure 5.1.



(a) 3D plot

(b) contour plot

Figure 5.1: Example of a two-dimensional Sphere benchmark. Optimal function value $f(\mathbf{x}^{opt})$ in optimum $\mathbf{x}^{opt} = (2.5, 2.5)$ is shown by a red arrow/cross.

5.1.2 Ellipsoidal Function

Ellipsoidal function represents unimodal, ill-conditioned continuous search problem with smooth local irregularities. Ill-conditioned function is a function such that a small change in the input vector of the function may lead to a large change in the resulting function value. Ellipsoidal function is defined as [20]:

$$\begin{aligned} f(x) &= \sum_{i=1}^{D} 10^{6\frac{i-1}{D-1}} z_i^2 + f_{opt} \\ \mathbf{z} &= T_{osz}(\mathbf{x} - \mathbf{x}^{opt}) \end{aligned} \tag{5.2}$$

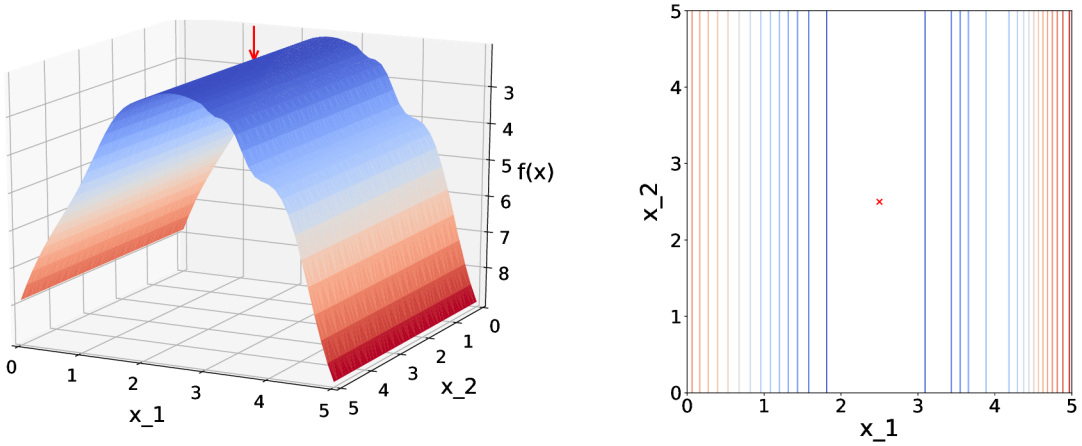
where $T_{osz} : \mathbb{R}^D \to \mathbb{R}^D$ is mapped element-wise for each element of the input vector and is defined as follows:

$$\begin{aligned} & x \mapsto \text{sign}(x)\exp(\hat{x} + 0.049(\sin(c_1\hat{x}) + \sin(c_2\hat{x}))) \\ & \hat{x} = \begin{cases} \log(|x|) & \text{if } x \neq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases} \\ & c_1 = \begin{cases} 10 & \text{if } x > 0 \\ 5.5 & \text{otherwise} \end{cases} \qquad c_2 = \begin{cases} 7.9 & \text{if } x > 0 \\ 3.1 & \text{otherwise} \end{cases} \end{aligned} \tag{5.3}$$

Example of two-dimensional Ellipsoidal function is shown in Figure 5.2.

5.2 Testing the Toolkit on Benchmarks

Each of the listed experiments on the before mentioned benchmark functions has been averaged from over 100 runs with different function optimum. The location of the optimum



(a) 3D plot

(b) contour plot

Figure 5.2: Example of a two-dimensional Ellipsoidal benchmark. Optimal function value $f(\mathbf{x}^{opt})$ in optimum $\mathbf{x}^{opt} = (2.5, 2.5)$ is shown by a red arrow/cross. As can be seen, the benchmark is ill-conditional only in dimension of x_1, while x_2 has only a small influence on the resulting function value.

was selected randomly with uniform distribution from a subset of available search space, so the optimum lies in different sector of the search space in each run. These measures should provide reasonable assessment in performance of all tested features of GPOP.

Each experiment was run on a benchmark with different dimensionality, where each dimension was optimised on interval $[0, 5]$. Based on the dimensionality of the benchmark, different number of samples and optimisation steps were used to comply with the size of the search space. The number of samples and the number of optimisation steps used for different D-dimensional benchmarks are shown in Table 5.1. All experiments used Matérn ARD 5/2 kernel, random DSS and EI AF, any additional settings or differences are described in particular experiments.

Table 5.1: Number of optimisation steps and number of testing samples used in D-dimensional benchmark experiments.

Configuration	1D	2D	3D	4D	5D
#steps	10	15	20	25	30
#samples	25	20^2	15^3	10^4	5^5

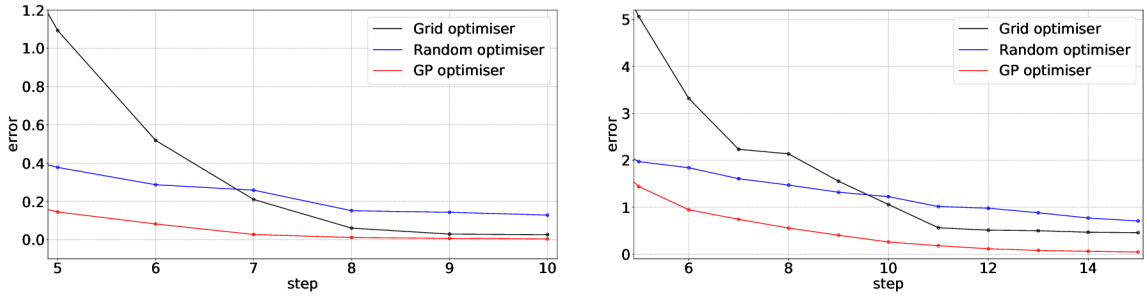
Experiments are divided into five sections, where first section focuses on comparison of implemented optimisers, next sections are focused on selected kernel, DSS and AF, while the last part focuses on automatic tuning of the GP parameters.

5.2.1 Optimiser Accuracy Comparison

The first set of experiments devotes attention to comparison between GP, grid and random hyper-optimisation. All of the optimisers were tested on up to 5-dimensional benchmarks. The accuracy of each optimiser is highly influenced by the size of search space and properties

of the benchmark function. In this set of experiments, GP optimiser with RBF kernel, random DSS and Expected Improvement AF was used.

Concerning smaller search space and sufficient amount of optimisation steps, Grid optimiser may provide better results than Random optimiser. But as the number of optimisation steps reduces or search space increases, Grid optimiser starts to provide worse results than GP optimiser or Random optimiser (see Figure 5.3).

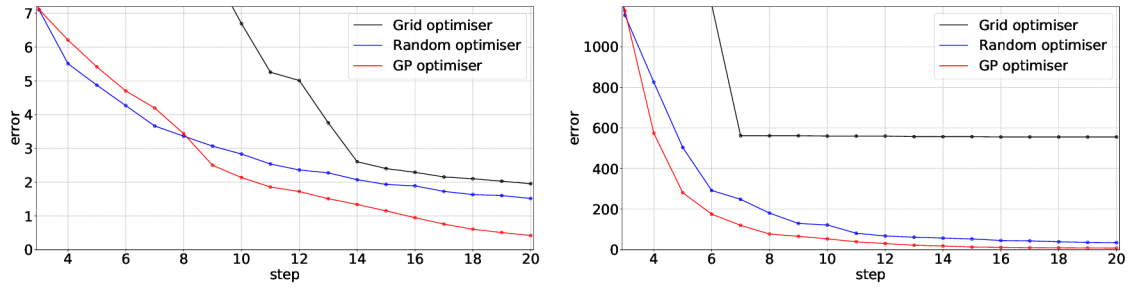


(a) 1D Sphere

(b) 2D Sphere

Figure 5.3: Comparison of GP, Random and Grid optimisers on 1D and 2D Sphere benchmarks showing the cumulative minimum over optimisation steps.

As can be seen in Figure 5.4, there is a mild difference between optimisation of Sphere and Ellipsoidal benchmark considering GP and random optimisation. Sphere benchmark is highly symmetrical and contains larger space with values closer to optimum. Therefore, Random optimiser tends to provide better results in the first few steps of optimisation (first 8 steps in Figure 5.4a) before GP optimiser creates sufficient model. Ellipsoidal benchmark is less symmetrical and ill-conditioned and therefore GP optimiser finds better solution faster, as can be seen in Figure 5.4b.

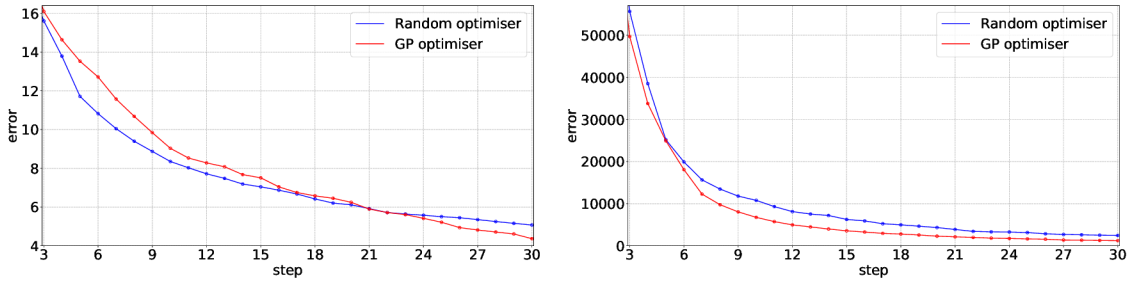


(a) 3D Sphere

(b) 3D Ellipsoidal

Figure 5.4: Comparison of GP, Random and Grid optimisers on 3D benchmarks showing the cumulative minimum over optimisation steps.

The number of steps needed for GP optimiser to beat average values of Random optimiser is influenced by the shape of the problem and by the size of the search space. When searching in larger or less symmetric space of a benchmark, the chances of random search to select better values are lesser. That means GP optimisation can achieve better results in just a few steps even when optimising problems with more dimensions, as can be seen in Figure 5.5.



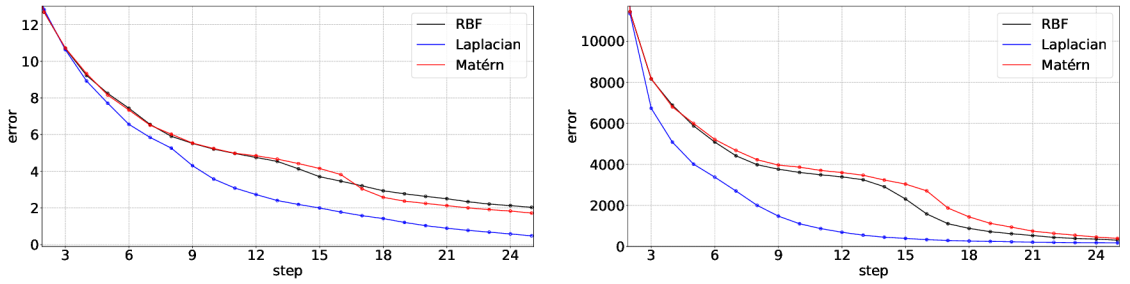
(a) 5D Sphere (b) 5D Ellipsoidal

Figure 5.5: Comparison of GP and Random optimiser on 5D benchmarks showing the cumulative minimum over optimisation steps. Random optimiser performs very-well on highly symmetrical Sphere benchmark and manages to provide better results for about 20 optimisation steps, while GP optimiser provides better optimisation results on Ellipsoidal benchmark.

5.2.2 Kernel Experiments

Kernel experiments were focused on performance of kernels themselves, without any differences between the parameters of the kernels. All three tested kernels (RBF, Laplacian and Matérn 5/2) used the same parameters setting in each conducted experiment, therefore Matérn kernel was used without ARD. Experiments were run on both before mentioned benchmarks in up to 5-dimensions.

As shown in Figure 5.6, kernel selection has a significant influence on the optimisation. The difference between the tested kernels on one and two-dimensional benchmarks is inconspicuous, but experiments on larger search space and in more dimensions show that Laplacian kernel suites these benchmarks better than the other two kernels. The performance of RBF and Matérn kernel is quite similar, but in most of the experiments RBF kernel performed slightly better. This is expected, since the behaviour of the kernels is quite similar and while Matérn is more suitable to model more substantial local changes, both used benchmarks are rather smooth.



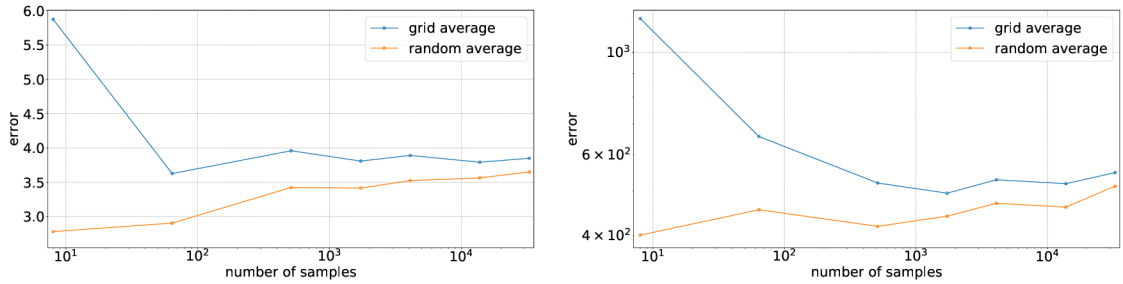
(a) 4D Sphere (b) 4D Ellipsoidal

Figure 5.6: Comparison of implemented kernels on 4D Sphere and Ellipsoidal benchmarks showing the cumulative minimum over optimisation steps. Kernel selection is highly dependent on the optimised problem and might reduce the number of optimisation steps quite significantly.

5.2.3 DSS Experiments

Next set of experiments was focused on implemented domain-space search strategies. The main objective of this set of experiments was to determine how different DSS strategies influence the result of hyper-optimisation and how are particular strategies influenced by used number of testing samples. Due to the high number of experiments needed to evaluate the latter, only benchmarks with up to 3-dimensions were tested, while the first set of experiments was run on up to 5-dimensional benchmarks.

When considering the influence of the number of samples on both DSS strategies, the optimal number of samples depends on the optimised benchmark, the number of optimisation steps, the number of dimensions and the size of the search space. As shown in Figure 5.7, the average performance of both grid and random DSS seem to share the same behaviour on each benchmark, except for smaller number of samples. All conducted DSS experiments in up to 5D show that starting from approximately 10^D samples (where D is the number of dimensions), the behaviour of both averages is very similar.

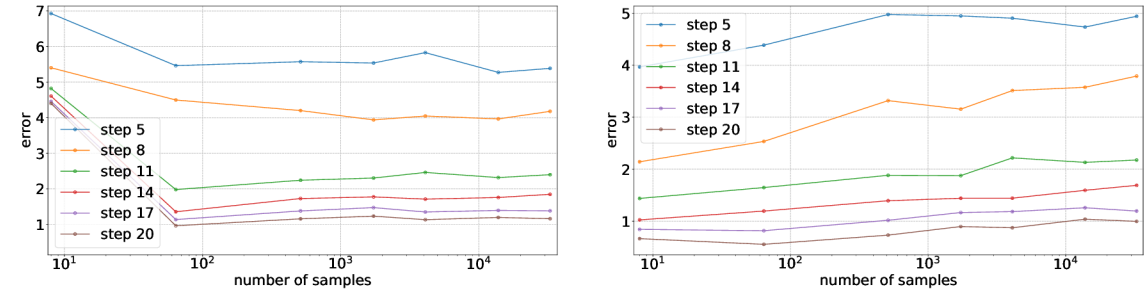


(a) 3D Sphere (b) 3D Ellipsoidal

Figure 5.7: Influence of the number of samples on average error of all the optimisation steps. The behaviour of both strategies is quite similar, except for very small number of samples. In comparison with grid DSS, random DSS achieves surprisingly good results even for a really small number of testing samples. However, this pattern isn't that striking in 1D experiments.

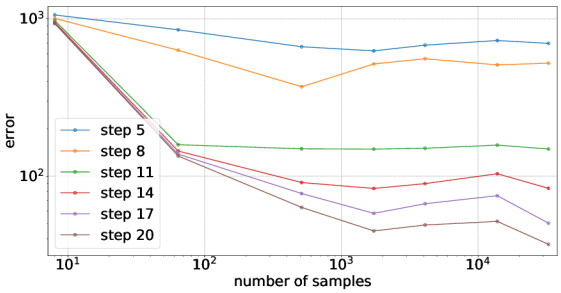
The average values serve as a good guideline for the comparison of a behaviour of both strategies, but they do not show the actual best achieved results. Figure 5.8 shows the best achieved results for selected optimisation steps in dependency on the number of samples. Because the results for each tested number of steps depends on so many factors, it's hard to select suitable number of testing samples. But it's obvious that grid DSS achieves poor results with small number of optimisation steps, while random DSS achieves more consistent results for every tested number of samples and moreover, it performs quite well even with a small number of testing samples.

Both random DSS and grid DSS behave analogously when changing the number of samples (except for very small values) and therefore experiments focusing on the influence of DSS strategies on minimizing the benchmark functions were run with the same amount of samples for both strategies. As results on Figure 5.9 suggest, hyper-optimisation with random DSS achieved better results then grid DSS in all conducted experiments. Furthermore, the difference between both approaches is more notable when solving benchmarks with more dimensions and experiments conducted on Ellipsoidal benchmark showed more substantial differences between tested DSS strategies.

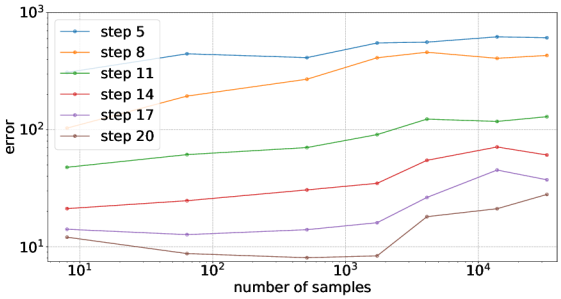


(a) grid DSS on 3D Sphere

(b) random DSS on 3D Sphere



(c) grid DSS on 3D Ellipsoidal



(d) random DSS on 3D Ellipsoidal

Figure 5.8: Influence of the number of samples on error in some of the optimisation steps. As can be seen in figures above, the best selected number of testing samples depends on many factors, such as selected DSS, number of optimisation steps and optimised benchmark.

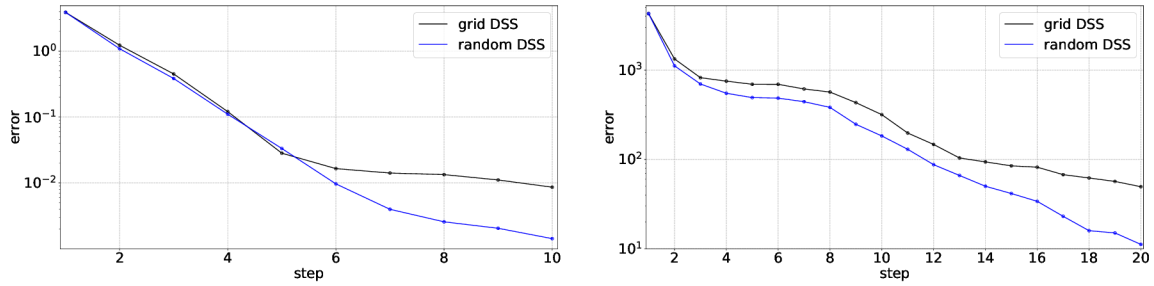
5.2.4 Comparison of Acquisition Functions

Next set of experiments compared three acquisition functions: `MinimalMean`, `LowerConfidence` and `ExpectedImprovement`. Experiments on one-dimensional benchmarks showed the best results achieved `LowerConfidence` AF, while the rest of the experiments (up to 5D for both benchmarks) showed better convergence to optimal value while using `ExpectedImprovement` AF. As shown in Figure 5.10, the difference between used AF was most perceptible in experiments with Ellipsoidal benchmark, while results on Sphere benchmark show only minimal difference between used AFs.

5.2.5 Estimation of GP Parameters

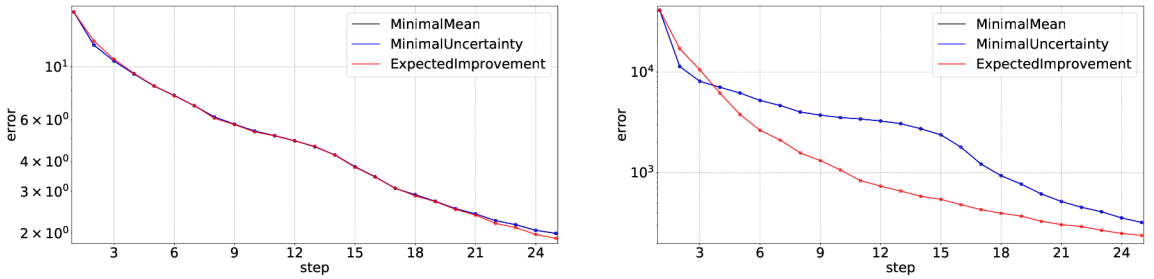
The last part of experiments on benchmarks was focused on automatic tuning of GP parameters. Three options were tested: GP optimisation without automatic tuning, with automatic tuning of kernel parameters and with automatic tuning of all parameters (kernel and uncertainty).

The best results on most of the benchmarks were achieved with automatic tuning of kernel parameters, as shown in Figure 5.11. Mentioned problem with tuning the uncertainty is probably caused because of the automatic tuning algorithm, which is trying to improve the log likelihood by increasing the uncertainty. That leads to the possibility of also increasing characteristic length scale, so these two parameters are increased until the model no longer fits the problem.



(a) 1D Sphere benchmark (b) 3D Ellipsoidal

Figure 5.9: Comparison of grid and random DSS.



(a) 4D Sphere (b) 4D Ellipsoidal

Figure 5.10: Influence of selected AF on the result of the optimisation. All AFs in experiments on Sphere benchmarks performed almost identically, except `ExpectedImprovement` AF was able to converge closer to the benchmark optimal function value, if given enough optimisation steps. This difference was more significant on Ellipsoidal benchmarks, where except first few optimisation steps, `ExpectedImprovement` AF achieved distinctly better results that the other two AFs (note that the results for `MinimalMean` and `LowerConfidence` shown in Figure b) coincide).

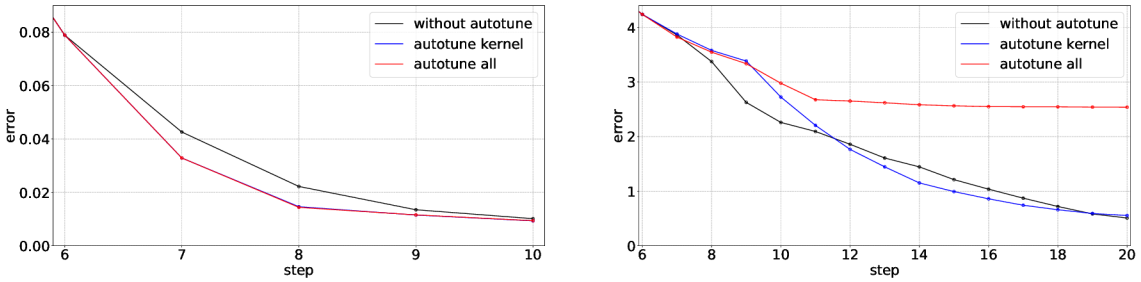
5.3 Neural Networks Experiments

NN experiments were run on MNIST dataset, specifically on image classification example NN[1]. NN in the example uses Stochastic Gradient Descent algorithm and trains the network for 10 epochs. All experiments compare three different optimisers: Grid optimiser, Random optimiser and GP optimiser. GP optimiser was run in two configurations, once with fixed parameters and once with automatic tuning of kernel parameters.

All optimised hyper-parameters and used dimension bounds are presented in Table 5.2. GP optimiser used random DSS, EI AF, uncertainty of 1e−3 and Matérn ARD 5/2 kernel with signal standard deviation $\sigma_f = 1.0$ and two distinct settings of characteristic length scale $\boldsymbol{\sigma}_l$, as defined in Table 5.3. Used number of optimisation steps and number of samples is the same as in benchmarks experiments and is described in Table 5.1.

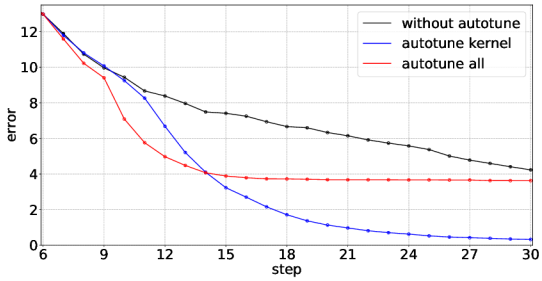
All results show cumulative minimum of validation loss in different optimisation steps. All loss values are averaged from only 5 distinct optimisation runs, due to a longer training time of the NN. This causes noticeable dispersion of the resulting loss values, but it should

[1] https://github.com/pytorch/examples/tree/master/mnist

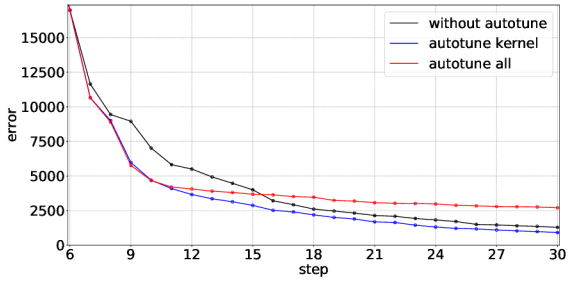


(a) Sphere 1D

(b) Sphere 3D



(c) Sphere 5D



(d) Ellipsoidal 5D

Figure 5.11: Influence of automatic tuning of GP parameters on a benchmark optimisation. One-dimensional benchmarks show only minor differences, but multidimensional benchmarks show that the automatic tuning of kernel parameters performs the best, while tuning of uncertainty leads in most cases to short improvement followed by significant deterioration of the optimisation. Though in one case the use of GP optimiser without automatic tuning achieved better results that GP optimiser with automatic tuning of kernel parameters, most experiments show that the latter yields significant improvement of the optimisation.

provide reasonable estimate to roughly compare the optimisers. Note that every experiment was run with two different settings of kernel parameter characteristic length scale $\boldsymbol{\sigma}_l$.

The aim of the first set of experiments was optimising a single hyper-parameter. As can be seen in Figure 5.12, Grid optimiser managed to find the best hyper-parameter setting of learning rate and number of hidden neurons. Grid optimiser searches the domain progressively, so the first and the last few optimisation steps search the border of a domain. That is the reason why the cumulative minimum of Grid optimiser usually changes rapidly in the beginning and maintains the value at the end of the optimisation. Random optimiser achieved the best result in optimisation of the batch size. But unlike Grid optimiser, its result is not influenced by the location of the optimum and the therefore the change in cumulative minimum is more gradual. GP optimiser needs a few steps before it creates

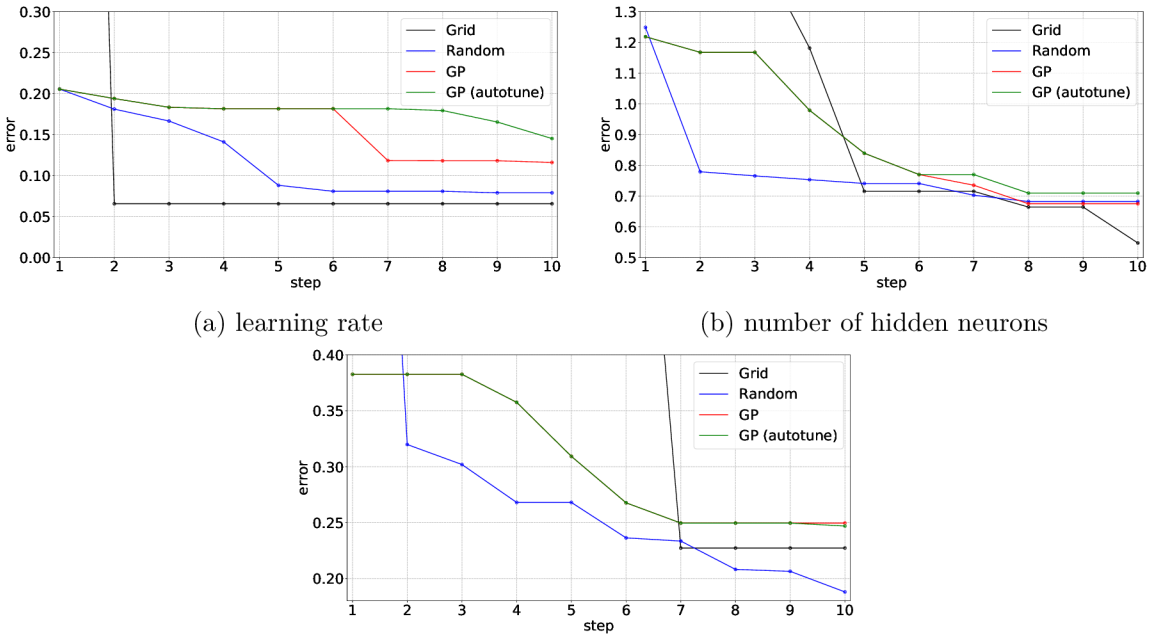
Table 5.2: Optimised hyper-parameters and their bounds used in experiments.

Hyper-parameter	Type	Dimension bounds
learning rate	float	$[0, 1]$
number of neurons in hidden layers	int	$[32, 512]$
batch size	int	$[1, 1024]$

Table 5.3: Characteristic length scale settings for optimised hyper-parameters used in experiments.

Hyper-parameter	σ_l	
	setting 1	setting 2
learning rate	0.1	0.2
number of hidden neurons	48	96
batch size	102	204

sufficient model and then starts to improve it's optimisation, but was not able to provide better results than the other optimisers.



(a) learning rate

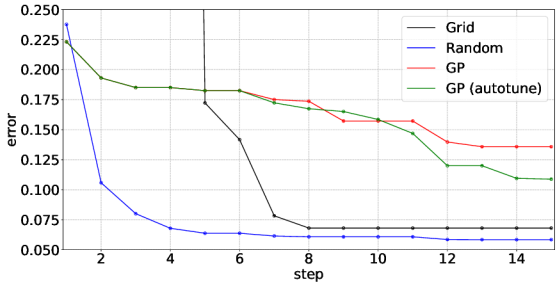
(b) number of hidden neurons

(c) batch size

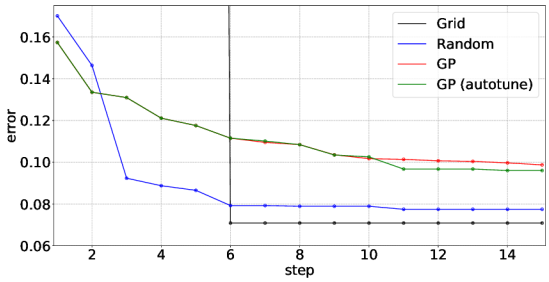
Figure 5.12: Comparison of Grid, Random and GP optimisers on MNIST NN, optimising one hyper-parameter.

The second set of experiments focused on optimisation of two hyper-parameters. As can be seen in Figure 5.13, Random optimiser achieved better result in the first few steps of optimisation, while Grid optimiser managed to find better values in two out of three experiments. GP optimiser performed similarly as in optimisation of one hyper-parameter, but there are more noticeable differences between GP optimiser with and without automatic parameter tuning. These differences are probably more noticeable because of higher number of optimisation steps with automatic parameter tuning.

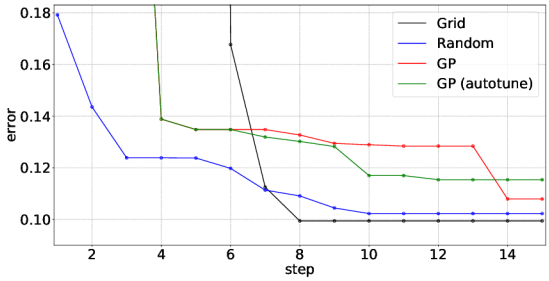
The last set of experiments was focused on optimisation of all three hyper-parameters. As can be seen in Figure 5.14, Random optimiser provided the best results throughout all the optimisation steps, while GP optimiser provided better results than Grid optimiser the first 13 steps. But this is influenced by the number of optimisation steps in total and Grid optimiser might possibly find better hyper-parameter settings even when using fewer



(a) learning rate, number of hidden neurons



(b) learning rate, batch size

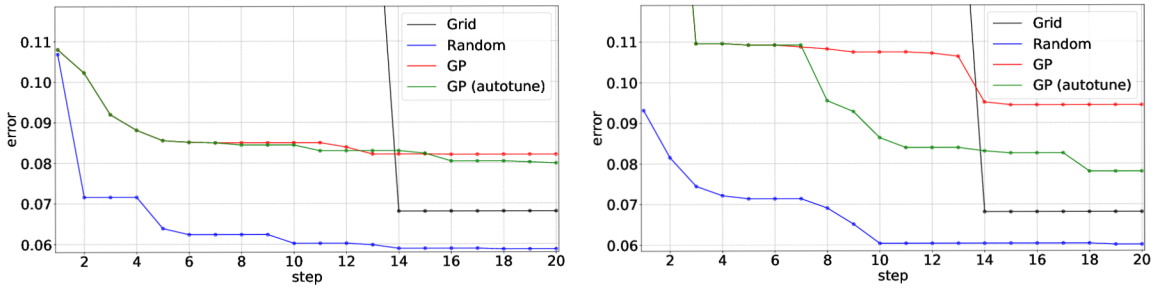


(c) number of hidden neurons, batch size

Figure 5.13: Comparison of Grid, Random and GP optimisation of two hyper-parameters of MNIST NN.

optimisation steps. Also, note that GP optimiser in Figure 5.14b managed to improve its results to the same loss values as in Figure 5.14a by using automatic parameter tuning.

The worse results of GP optimiser might be caused by several issues: GP parameter settings, inconvenient mean or the nature of the loss space. To fix parameters of the GP model, more knowledge about the behaviour of the optimised hyper-parameters is needed. Problems with mean could arise when the loss values are too close to mean value and GP optimiser might get stuck in one place. This issue could be resolved by selection of a different AF or a loss function. Loss space could cause problems when the domains of the optimised hyper-parameters are quite large and the characteristic length scale parameter is set to higher values. That could lead to inability to model the subtle differences in the loss value.



(a) $\boldsymbol{\sigma}_l = (0.1, 48, 102)$ (b) $\boldsymbol{\sigma}_l = (0.2, 96, 24)$

Figure 5.14: Comparison of Grid, Random and GP optimisers on MNIST NN, optimising three hyper-parameters: learning rate, number of hidden neurons and batch size. The results for two different characteristic length scale settings are shown. Note that both runs used the same Grid and Random optimiser, the differences in Random optimiser are caused only by different seed.

Chapter 6

Conclusion

The goal of this thesis was to design and implement a hyper-optimizer based on Gaussian Processes. I have achieved this goal by implementing a simple GP based hyper-optimisation library with a CLI wrapper. Furthermore, I compared the efficiency of implemented GP optimiser with two baseline solutions based on grid and random search by numerous experiments on a few benchmark functions and MNIST dataset. Also, I have tested different parameters of the GP optimiser, such as DSS or AF, to evaluate their influence on the result of the optimisation.

The experiments on the benchmarks functions proved that the implemented GP optimiser is in analogous cases able to achieve better results than both grid and random search optimisation techniques and in some cases may save more that ten optimisation steps. Experiments with parameters of the GP optimiser show that best option for DSS is random DSS. Optimisers with this strategy achieved better result in all tested cases and since random DSS provides better results with fewer testing samples, it also brings notable improvement in computation time of GP predictions. Most suitable AF proved to be EI, while most suitable kernel on the tested benchmarks was Laplacian kernel. Both EI AF and Laplacian kernel led to faster improvement, especially in the first few steps of the optimisation. Experiments with automatic tuning of the parameters have shown that tuning only the parameters of the kernel leads mostly to better results, while tuning kernel parameters with uncertainty leads to fast deterioration of the optimisation results in most cases.

The experiments on MNIST dataset show that average loss value achieved in random optimisation is better than in GP optimisation. Though the results of the GP optimiser can be improved by changing its parameters, it requires better understanding of behaviour of the optimised hyper-parameters.

In the future, I would like to continue my work and improve the performance and user interface of the implemented toolkit. Specifically, I would like to improve automatic tuning of GP parameters in contrast to their domain size. Also, I would like to further simplify the way of defining configuration files of neural networks and the addition of custom benchmark functions. My work could be further expanded by improving the performance of GP optimisation, adding additional hyper-optimisation methods or by creating a GUI for visualisation of the optimised hyper-parameters.

Bibliography

[1] Alto, V. *Neural Networks: parameters, hyperparameters and optimization strategies* [online]. Towards Data Science, july 2019 [cit. 2020-01-20]. Available at: https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5.

[2] Bergstra, J., Bardenet, R., Bengio, Y. and Kégl, B. Algorithms for Hyper-Parameter Optimization. In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2011, p. 2546–2554. NIPS'11. ISBN 9781618395993.

[3] Bergstra, J. and Bengio, Y. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* JMLR.org. february 2012, vol. 13, p. 281–305. Available at: http://dl.acm.org/citation.cfm?id=2188385.2188395. ISSN 1532-4435.

[4] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

[5] Bjorck, J., Gomes, C. P. and Selman, B. Understanding Batch Normalization. *CoRR*. 2018, abs/1806.02375. Available at: http://arxiv.org/abs/1806.02375.

[6] Brochu, E., Brochu, T. and Freitas, N. A Bayesian Interactive Optimization Approach to Procedural Animation Design. In:. July 2010, p. 103–112.

[7] Brochu, E., Cora, V. M. and Freitas, N. de. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *CoRR*. 2010, abs/1012.2599. Available at: http://arxiv.org/abs/1012.2599.

[8] Brochu, E., Hoffman, M. and Freitas, N. de. *Hedging Strategies for Bayesian Optimization*. ArXiv:1009.5419. Sep 2010. Available at: http://cds.cern.ch/record/1295294.

[9] Bushaev, V. *How do we 'train' neural networks?* [online]. Towards Data Science, 2017 [cit. 2020-05-02]. Available at: https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73.

[10] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B. and LeCun, Y. The Loss Surface of Multilayer Networks. *CoRR*. 2014, abs/1412.0233. Available at: http://arxiv.org/abs/1412.0233.

[11] Clevert, D.-A., Unterthiner, T. and Hochreiter, S. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In:. January 2016.

[12] Dewancker, I., McCourt, M. and Clark, S. *Bayesian optimization primer*. 2015.

[13] Eggensperger, K. Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters. In:. 2013.

[14] Eggensperger, K., Hutter, F., Hoos, H. H. and Leyton Brown, K. Surrogate Benchmarks for Hyperparameter Optimization. In: *Proceedings of the 2014 International Conference on Meta-Learning and Algorithm Selection - Volume 1201*. Aachen, DEU: CEUR-WS.org, 2014, p. 24–31. MLAS'14. ISBN 16130073.

[15] Fogel, D. B., Fogel, L. J. and Porto, V. W. Evolutionary programming for training neural networks. In: *1990 IJCNN International Joint Conference on Neural Networks*. 1990, p. 601–605 vol. 1.

[16] Gaurang, P., Ganatra, A., Kosta, Y. and Panchal, D. Behaviour Analysis of Multilayer Perceptronswith Multiple Hidden Neurons and Hidden Layers. *International Journal of Computer Theory and Engineering*. january 2011, vol. 3, p. 332–337.

[17] Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J. E. et al., ed. *Google Vizier: A Service for Black-Box Optimization*. 2017. Available at: http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization.

[18] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618.

[19] Görtler, J., Kehlbeck, R. and Deussen, O. A Visual Exploration of Gaussian Processes. *Distill*. 2019. https://distill.pub/2019/visual-exploration-gaussian-processes.

[20] Hansen, N., Ros, R. and Auger, A. Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. In:. 2009.

[21] Hutter, F., Lücke, J. and Schmidt Thieme, L. Beyond Manual Tuning of Hyperparameters. *KI - Künstliche Intelligenz*. july 2015, vol. 29.

[22] Koehrsen, W. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning* [online]. Towards Data Science, 2018 [cit. 2020-05-05]. Available at: https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f.

[23] Lecun, Y. A Theoretical Framework for Back-Propagation. august 2001.

[24] LeCun, Y., Bottou, L., Orr, G. B. and Müller, K.-R. Efficient BackProp. In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, p. 9–50. ISBN 3540653112.

[25] LeCun, Y. and Cortes, C. MNIST handwritten digit database. [http://yann.lecun.com/exdb/mnist/]. 2010. Available at: http://yann.lecun.com/exdb/mnist/.

[26] Liaw, A., Wiener, M. et al. Classification and regression by randomForest. *R news*. 2002, vol. 2, no. 3, p. 18–22.

[27] Lu, L., Shin, Y., Su, Y. and Karniadakis, G. E. Dying ReLU and Initialization: Theory and Numerical Examples. *ArXiv*. 2019, abs/1903.06733.

[28] Matuszyk, P., Castillo, R. T., Kottke, D. and Spiliopoulou, M. A Comparative Study on Hyperparameter Optimization for Recommender Systems. In: Lex, E., Kern, R., Felfernig, A., Jack, K., Kowald, D. et al., ed. *Workshop on Recommender Systems and Big Data Analytics (RS-BDA'16) @ iKNOW 2016*. 2016. Available at: http://socialcomputing.know-center.tugraz.at/rs-bda/.

[29] Olof, S. S. A Comparative Study of Black-box Optimization Algorithms for Tuning of Hyper-parameters in Deep Neural Networks. In:. 2018.

[30] Ramachandran, P., Zoph, B. and Le, Q. V. Searching for Activation Functions. *CoRR*. 2017, abs/1710.05941. Available at: http://arxiv.org/abs/1710.05941.

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

[32] Ruder, S. An overview of gradient descent optimization algorithms. *ArXiv preprint arXiv:1609.04747*. 2016.

[33] Sexton, R., Dorsey, R. and Johnson, J. Beyond backpropagation: Using simulated annealing for training neural networks. *Journal of End User Computing*. july 1999, vol. 11.

[34] Snoek, J., Larochelle, H. and Adams, R. P. Practical Bayesian Optimization of Machine Learning Algorithms. In: Pereira, F., Burges, C. J. C., Bottou, L. and Weinberger, K. Q., ed. *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, p. 2951–2959. Available at: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

[35] Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N. et al. Scalable Bayesian Optimization Using Deep Neural Networks. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. JMLR.org, 2015, p. 2171–2180. ICML'15.

[36] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 2014, vol. 15, no. 56, p. 1929–1958. Available at: http://jmlr.org/papers/v15/srivastava14a.html.

[37] Vasani, D. *This thing called Weight Decay* [online]. Towards Data Science, 2019 [cit. 2020-05-11]. Available at: https://towardsdatascience.com/this-thing-called-weight-decay-a7cd4bcfccab.