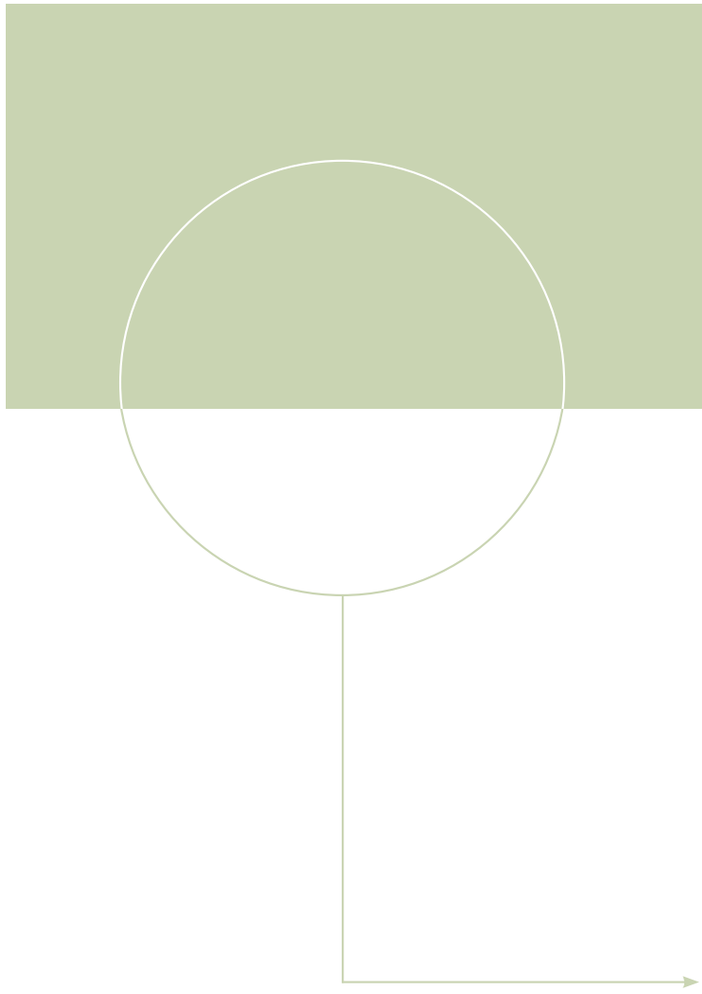


Master's thesis

2020

Bjørn Magnus Valberg Iversen

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

Bjørn Magnus Valberg Iversen

Combining Hyperband and Gaussian Process-based Bayesian Optimization

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Combining Hyperband and Gaussian Process-based Bayesian Optimization

Bjørn Magnus Valberg Iversen

Computer Science
Submission date: June 2020
Supervisor: Magnus Lie Hetland

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Identifying good hyperparameters for machine learning models is an important task. So important, in fact, that an entire field is dedicated to this very task—this field is called hyperparameter optimization. There are many hyperparameter optimization algorithms, such as grid search, random search and evolutionary algorithms, to name a few. This thesis examines a combination of two algorithms—these algorithms being Bayesian optimization and Hyperband. The thesis shows how Gaussian process-based Bayesian optimization and Hyperband can be combined through Bayesian sampling, and tests a new combination of these methods based on previous work. The experimental results indicate that the combined approach should not be preferred over vanilla Hyperband, but reasons as to why this might be the case and ideas for future work are presented.

Sammendrag

Å identifisere gode hyperparametre for maskinlæringsmodeller er en viktig oppgave. Denne oppgaven er såpass viktig at den har inspirert et eget felt, nemlig hyperparameteroptimering. Det finnes mange hyperparameteroptimeringsalgoritmer, som nettsøk, tilfeldig søk og evolusjonære algoritmer, bare for å nevne noen. I denne masteroppgaven undersøkes kombinasjonen av to ulike hyperparameteroptimeringsalgoritmer – Bayesisk optimering og Hyperband. Oppgaven viser hvordan Bayesisk optimering basert på Gaussiske prosesser og Hyperband kan kombineres gjennom Bayesisk sampling, og tester en ny kombinasjon av disse basert på tidligere arbeid gjennom eksperimenter. De eksperimentelle resultatene tilsier at kombinasjonen av metodene ikke er å foretrekke over vanlig Hyperband, men mulige grunner for dette skisseres, og mulige endringer for fremtidig arbeid foreslås.

Acknowledgements

By the time this thesis has been delivered, my life as a student has concluded. That fact is hard to grasp. It is, however, as many things in life, inevitable. Nevertheless, I am looking forward to new challenges elsewhere.

I would like to thank my supervisor, Magnus Lie Hetland, for answering all my questions no matter when, and for always being supportive.

If it had not been for my great friends, I don't think I would have persevered for as long as I have. You all mean a lot to me, even if I seldomly say so explicitly. Thank you. An especially big thanks goes to Ane, my girlfriend, for providing all the emotional support in the world.

Finally, a big thank you to my parents for always supporting me no matter what.

Contents

Acronyms

ANN Artificial Neural Network. 10

BO Bayesian Optimization. 16

CNN Convolutional Neural Network. 13

GP Gaussian Process. 17

HB Hyperband. 28

ReLU Rectified Linear Unit. 11

SVM Support Vector Machine. 6

Chapter 1

Introduction

1.1 Motivation and Goal

Machine learning as an area of research has gained tremendous traction in the recent years. Machine learning algorithms take as input a set of data $\mathcal{D} = \{x_i, y_i\}_{i=1}^n$ and a set of hyperparameters θ, and attempt to produce mappings $f : \mathcal{X} \to \mathcal{Y}$ such that $f(x_i) \approx y_i$—these mappings are referred to as *models*. As an example of a machine learning problem, each *feature* x_i could be an English word, and each *label* y_i could be a boolean that indicates whether the corresponding word is a noun. The task of the machine learning algorithm is then to produce a model that can differentiate words that are nouns from words that are not nouns. In this example of a machine learning problem, since each label y_i is discrete-valued, the problem is referred to as a *classification problem*; in the case that each label y_i is real-valued, the problem is referred to as a *regression problem*.

Crucial to the quality of the model produced by any machine learning algorithm is the quality, diversity and size of the available data pool, but also the setting of the hyperparameters θ. It is the search for these hyperparameters, an area of research referred to as *hyperparamter optimization*, that this thesis is concerned with. To gauge the quality of a particular configuration of hyperparameters, it is customary to split the data into two sets of data, referred to as the *training data set* and the *test data set*. The machine learning algorithm is then ran to produce a model from the training data set and the hyperparameter configuration selected. Once the model is produced, the average *loss* of the

model is calculated in light of the test data set—the lower the loss, the better the hyperparameter configuration. The data set is split in this manner to avoid the problem of data set bias—the average loss of the model over the training data set is a poor measure of generalization ability, since the model is biased towards that particular data set.

One of the most unsophisticated ways to search through the space of hyperparameters is to perform a *grid search*. In a grid search, the hyperparameter subdomain of interest, typically a hypercube, is partitioned into a multi-dimensional grid. Each point of intersection in the grid defines a hyperparameter configuration to try, and the point with the lowest average loss is returned as the best hyperparameter configuration. Simple to implement and understand, grid search is a popular choice among machine learning practitioners, but as noted by J. Bergstra and Bengio (2012), *random search* is often a much better choice. Random search works by simply picking each hyperparameter at random from an appropriate prior probability distribution. The reason random search seems to outperform grid search is because many of the hyperparameters are unimportant. In a grid search, the hyperparameters are searched in a dependent manner—for each value of an unimportant hyperparameter, every combination of the remaining hyperparameters is tried. In a random search, the hyperparameters are searched in an independent manner, thus avoiding the problem of unimportant hyperparameters.

Training models and calculating average test losses are extremely time-consuming procedures. A hyperparameter optimization framework that attempts to minimize the number of these procedure calls is *Bayesian optimization* (Shahriari et al. 2015), which can be used to optimize noisy black-box functions $f(x)$. The way Bayesian optimization works is that it maintains a *surrogate* of f—a popular choice for this surrogate is the Gaussian Process (Rasmussen 2003). This surrogate models the posterior of f given x, $p(f \mid x)$, and is often orders-of-magnitude less expensive to evaluate than f itself. In each iteration of a Bayesian optimization procedure, an *acquisition function*, which depends on the surrogate, is used to propose the next point x_{next} to evaluate f. This new pair $(x_{next}, f(x_{next}))$, along with the entire evaluation history of f, is then used to update the surrogate. This Propose-Evaluate-Update cycle, shown in Figure 1.1, is then repeated until the computational budget is exhausted, or a satisfactory point x^* is found. In the case of hyperparameter optimization, the function f to optimize is the average test loss of the model, and each point x corresponds to a hyperparameter configuration.

Orthogonal to the Bayesian approach is an algorithm known as Hyperband (Li et al. 2017). Hyperband works by iteratively providing more and more

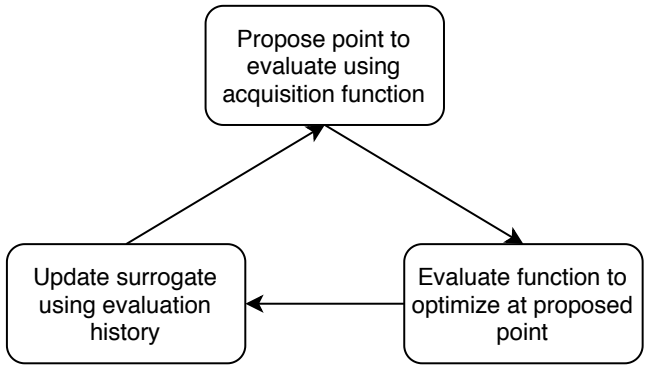


Figure 1.1: The Propose-Evaluate-Update cycle of the Bayesian optimization framework.

resources to hyperparameter configurations that seem promising. A resource could for instance be the number of epochs to train an artificial neural network. It uses random search to sample hyperparameter configurations, and has been shown empirically to provide significant speedups over both random search and Bayesian optimization.

Since Bayesian optimization and Hyperband are two orthogonal approaches—orthogonal in the sense that they approach the same problem differently—why not try to combine them? This question leads to the goal of this thesis: **To determine whether Hyperband could be improved by combining it with Bayesian Optimization**. Falkner, Klein, and Hutter (2018) combine Hyperband and Bayesian Optimizatio in an algorithm they dub BOHB—this algorithm utilizes a surrogate known as TPE. Motivated by this fact, and the fact that Gaussian Processes are the most common surrogates, I want to use a Gaussian Process as surrogate in my thesis.

Combining Hyperband with Bayesian optimization using a Gaussian process as surrogate is not a new idea. Bertrand et al. (2017) combine Hyperband with Bayesian optimization using a Gaussian process surrogate, but their work has a few flaws. First of all, as the authors themselves state, their method lacks an explicit notion of resource that Hyperband uses, and how to exploit it. Secondly, they only performed a single experiment, which is not enough to draw any statistically significant conclusions.

1.2 Research Questions

To reach the goal of the thesis, which is to determine whether Hyperband can be improved by combining it with Gaussian process-based Bayesian optimization, there are a few questions that naturally arise. The most obvious of which is probably *how*, as in how can the two approaches be combined? And after the fact, how can the methods be compared? Assuming that there is one optimal hyperparameter configuration $\hat{\theta}$ with corresponding loss $\hat{L}$, the methods can be compared by observing how fast they approach $\hat{L}$. The method that approaches $\hat{L}$ the quickest is deemed superior. Since $\hat{L}$ is generally not known *a priori*, we can only compare the results produced by each method side by side. The preceding questions and observations lead to the following research questions:

- **RQ1**: How can Hyperband and Gaussian Process-based Bayesian Optimization be combined into *Bayesian Hyperband*?
- **RQ2**: Does Bayesian Hyperband improve minimum error convergence rate compared to standard Hyperband?

1.3 Contributions

The main contribution of this thesis is the presentation and empirical evaluation of the method in chapter 5 that is inspired by Falkner, Klein, and Hutter (2018) and Bertrand et al. (2017).

1.4 Thesis Structure

The thesis is separated into two segments that have different purposes. The first segment, consisting of chapters 2, 3 and 4, presents the background theory necessary to understand the thesis. The second segment, consisting of chapters 5, 6 and 7, presents and empirically evaluates a method that combines Hyperband and Gaussian process-based Bayesian optimization.

Chapter 2 presents the machine learning algorithms known as Support Vector Machines, Artificial Neural Networks and Convolutional Neural Networks; chapter 3 presents Bayesian optimization with a focus on the Gaussian process as surrogate; chapter 4 presents the hyperparameter optimization algorithm known as Hyperband; chapter 5 presents a method that attempts to combine Gaussian process-based Bayesian optimization with Hyperband; chapter 6 shows some

experiments that compare the hybrid approach with standard Hyperband; and finally, chapter 7 addresses the research questions posed in section 1.2.

Chapter 2

Machine Learning Algorithms

This section aims to provide conceptual explanations of the machine learning algorithms used in chapter 6. Consequently, many details will be omitted. References to appropriate articles will be given where applicable.

2.1 Support Vector Machines

Support Vector Machines (or SVMs for short) are, in their simplest form, binary linear classifiers. An SVM performs classification by constructing a hyperplane $\vec{w} \cdot \vec{x} - b = 0$ where $\vec{w}$ is the normal vector, and each point $\vec{x}$ is assigned a class depending on which side of the hyperplane it is located.

This subsection explains SVMs to some level of detail, but for a more thorough description, consult the article by Cortes and Vapnik (1995).

2.1.1 Hard-margin

Imagine now that we are given a set of n linearly separable training points $(\vec{x_1}, y_1), ..., (\vec{x_n}, y_n)$, where $\vec{x}_i \in \mathbb{R}^d$, $y_i \in \{-1, 1\}$, and we wish to train an SVM on these data points. The goal of the SVM is to construct a hyperplane $\vec{w} \cdot \vec{x} - b = 0$ that cleanly separates the training data into two regions, each region containing all the points that belong to either class. Since the data is linearly separable, there are infinitely many such hyperplanes to choose from. SVMs

have an additional constraint, namely that the distance from the hyperplane to the closest point belonging to either class is maximized.

To find the hyperplane that maximizes the distance to the closest point of either class, the *maximum margin* hyperplane, we can construct one decision boundary for each class, and maximize the distance between the two boundaries—the hyperplane of interest will then be right in the middle. See section 2.1.1 for a visual depiction. We define the first boundary as $\vec{w}\cdot\vec{x}-b=1$—any point on or above this boundary is classified as a 1. We define the second boundary as $\vec{w}\cdot\vec{x}-b=-1$—any point on or below this boundary is classified as a -1. The hyperplane right in the middle between the two boundaries is then $\vec{w}\cdot\vec{x}-b=0$, and the objective is to maximize the distance between the boundaries. This distance can be shown to be $\frac{2}{||\vec{w}||}$, so the problem can be stated as minimizing $||\vec{w}||$. Additionally, since the training data should respect the boundaries, the objective is constrained by $\vec{w}\cdot\vec{x_i}-b\geq 1$ if $y_i=1$ or $\vec{w}\cdot\vec{x_i}-b\leq -1$ if $y_i=-1$ for each training point. Note that only the points that lie on their respective boundaries determine the width of the margin—these points are called support vectors. Rewriting the constraints in a neater form, the optimization problem becomes ***minimize*** $\frac{2}{||\vec{w}||}$ ***subject to*** $y_i(\vec{w}\cdot\vec{x_i}-b)\geq 1\ \ \forall\ i\in\{1,...,n\}$. The $\vec{w}$ and b that solve this problem determine the classifier $\text{sgn}(\vec{w}\cdot\vec{x}-b)$.

2.1.2 Soft-margin

What do we do if the training data isn't linearly separable? One solution is to alter the optimization problem. First, let's introduce the hinge loss function $\max(0,1-y_i(\vec{w}\cdot\vec{x_i}-b))$. If the constraint for the ith training point as defined in section 2.1.1 is satisfied, then the value of this function is 0. If the constraint is not satisfied however, the value of the function is greater than 0, and increases as x_i moves further away from the correct region. Incorporating the hinge loss function, the new optimization problem becomes **minimize**

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max(0,1-y_i(\vec{w}\cdot\vec{x_i}-b))\right]+\lambda||w||^2. \tag{2.1}$$

In eq. (2.1), λ is a user-defined parameter that controls the trade-off between misclassification and the width of the margin. As $\lim_{\lambda\to\infty}$, only the width of the margin is emphasized, but as $\lim_{\lambda\to 0}$, the method essentially turns into the one described in section 2.1.1. The choice of λ can greatly impact the generalization ability of the resulting classifier, and should therefore be tuned carefully.

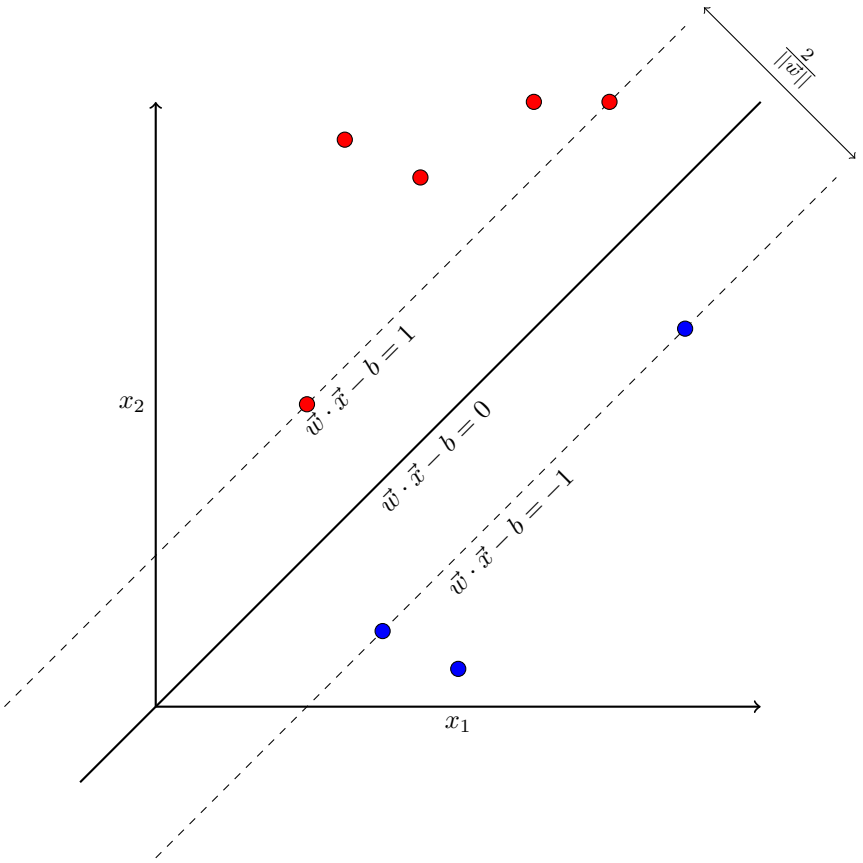


Figure 2.1: A depiction of how a hard-margin SVM finds the maximum-margin hyperplane. The red points belong to class 1, while the blue points belong to class -1. The width of the margin is maximized while ensuring that the data points respect their boundaries. The maximum-margin hyperplane is the hyperplane midway between the two boundaries (the dashed lines in the figure).

2.1.3 Kernel Trick

Nothing stops us from mapping each input $\vec{x}_i$ to a higher-dimensional space to deal with data sets that are not linearly separable. With such a mapping $\phi(\vec{x})$, we could simply find the right hyperplane in the transformed space. Un-

fortunately, such transformations are often hard to describe explicitly. Luckily, there is a solution, which is often referred to as *the kernel trick* in the literature. Instead of having explicit knowledge of $\phi(\vec{x})$, we only need a kernel function $k(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$ to construct the classifier in the higher-dimensional space. Thus, we only need to know the dot product between $\phi(\vec{x}_i)$ and $\phi(\vec{x}_j)$. Several kernels have been proposed, and one of the most popular ones is the Radial Basis Function (RBF) kernel, which is defined as

$$k(\vec{x}_i, \vec{x}_j) = \exp\big(-\gamma||\vec{x}_i - \vec{x}_j||\big), \tag{2.2}$$

where γ is a free parameter.

2.1.4 Multi-label Classification

The simplest strategy to perform multi-label classification with SVMs is called *one-versus-all*. In this scheme, if there are N classes, N separate SVMs are trained on the data. Each SVM is assigned a class, and learns to classify instances as belonging to that class or not. To combine the N SVMs on a new instance, the SVM with the largest output on that instance assigns the class.

Another strategy is *one-versus-one*. In this scheme, $\frac{N(N-1)}{2}$ SVMs are trained, one for each pair of labels. To classify a new instance, each SVM votes on which class it is. The class with the most votes is the assigned class. While the training time of this approach scales quadratically with the number of labels, it is still the most popular one, and is used in libraries such as Scikit-learn (Pedregosa et al. 2011).

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs, also referred to as simply *neural networks*) are biologically inspired, layered structures mainly used for supervised learning (Goodfellow, Bengio, and Courville 2016). Abstractly, ANNs accept an input $\boldsymbol{x} = (x_1, x_2, ..., x_m)$ and produce an output $\boldsymbol{y} = (y_1, y_2, ..., y_n)$—in other words, an ANN is simply a mapping $f : \mathbb{R}^m \to \mathbb{R}^n$. Concretely, these mappings are imposed by the interplay between the building blocks of the ANN, of which *neurons* and *weights* are the most fundamental.

Figure 2.2 shows a simple neural network containing six neurons distributed across three layers. The neural network in the figure has two input neurons in its *input layer* (neuron 0 and 1), each receiving its own input, and two output neurons in its *output layer* (neuron 4 and 5), each outputting a single value. Layers between the input and output layers of a neural network are referred to as *hidden layers*. The neural network in the figure contains a single hidden layer consisting of neuron 2 and 3. Note that the neurons in each layer receive inputs from all neurons in the previous layer as indicated by the directed edges, barring the input layer; this kind of connectivity is a feature of neural networks.

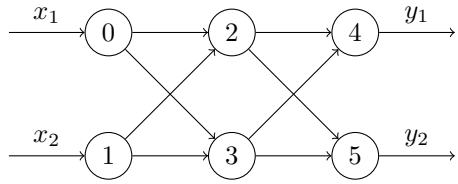


Figure 2.2: A sample neural network with one hidden layer.

When an input $\boldsymbol{x}$ is presented to a neural network, the neurons of the network fire in turn, layer by layer, to produce a final output $\boldsymbol{y}$. Each neuron j receives a signal from every neuron i in the preceeding layer that is weighted by some weight w_{ij}, and the output of neuron j is defined as

$$o_j = \phi\left(\sum w_{ij} o_i\right), \tag{2.3}$$

where $\phi(x)$ is an *activation function*, o_i is the output from neuron i, and w_{ij} is the weight between neuron i and j. The activation function determines the strength of a neuron's signal, and aims to add non-linearity to the neural network. There are many different activation functions, and there is usually a single

activation function associated with each layer. A popular choice of activation function for the hidden layers is the Rectified Linear Unit (ReLU) (Nair and G. E. Hinton 2010)

$$\phi(x) = \max(0, x), \tag{2.4}$$

which is depicted in fig. 2.3, and a popular choice for the output layer in case of classification is Softmax (Bishop 2006), where the output of neuron k is defined to be

$$o_k = \frac{\exp(in_k)}{\sum_{i=1}^{m} \exp(in_i)}, \tag{2.5}$$

where in_k is the weighted sum of inputs to neuron k and m is the number of neurons in the layer.

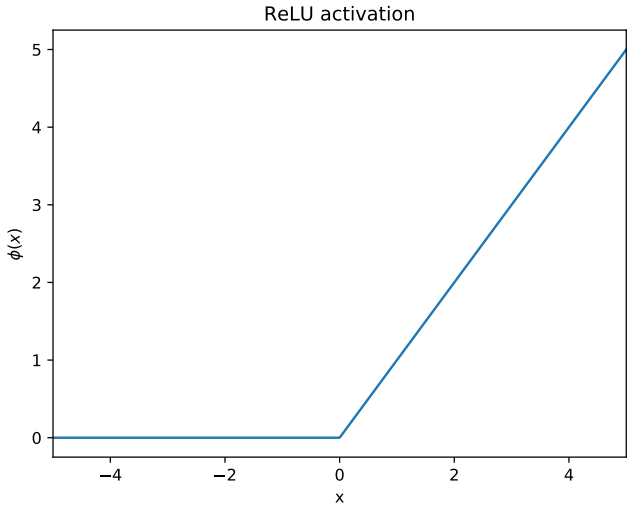


Figure 2.3: The ReLU activation function.

Sometimes we may wish to translate the activation function along the x-axis. This can be done adding a *bias term*, so that the output of neuron j becomes

$$o_j = \phi\left(\left[\sum w_{ij} o_i\right] + b_j\right), \tag{2.6}$$

where b_j is the bias associated with neuron j. The bias term can be thought of as a neuron that outputs 1 with weight b_j.

2.2.1 Tuning the Weights

Suppose we have a pool of training data consisting of input-output pairs of the form $(\boldsymbol{x}, \boldsymbol{t})$. How should the weights of an arbitrary neural network be set to best fit the data? We first need a way to measure how well the network fits the data. This is where the *loss function* comes in. The loss function L takes as input the target value $\boldsymbol{t} = (t_1, ..., t_n)$ and the value $\boldsymbol{y} = (y_1, ..., y_n)$ computed by the network, and returns a real number. The lower the number returned by L, the better the predicted value $\boldsymbol{y}$ approximates the target $\boldsymbol{t}$. A common loss function for regression problems is the *mean squared error*, which is defined as

$$L(\boldsymbol{y}, \boldsymbol{t}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - t_i)^2, \tag{2.7}$$

where n is the number of components in $\boldsymbol{y}$ and $\boldsymbol{t}$. A common loss function for classification problems is the logarithmic loss, defined to be

$$L(\boldsymbol{y}, \boldsymbol{t}) = \sum_{i=1}^{n} t_i \log(y_i), \tag{2.8}$$

where it is assumed that $t_i = 1$ if the associated input $\boldsymbol{x}$ belongs to class i and 0 otherwise.

What we want to do is to find a weight-vector $\boldsymbol{w}$ that minimizes the average loss across the data set, i.e. $L_{\text{avg}} = \frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{y}_i, \boldsymbol{t}_i)$, where m is the number of training examples. Note that if we denote the output of the network for an input $\boldsymbol{x}_j$ as $f(\boldsymbol{x}_j)$, then $L(\boldsymbol{y}_j, \boldsymbol{t}_j) = L(f(\boldsymbol{x}_j), \boldsymbol{t}_j)$, and so the loss at a single example is a function of the weights w_{ij} of the network. To find the minimizer $\boldsymbol{w}$, gradient descent is usually performed. Gradient descent is an iterative procedure, and in each iteration i the weight-vector is updated according to the rule $\boldsymbol{w}^{i+1} = \boldsymbol{w}^i - \eta \cdot \nabla L_{\text{avg}}(\boldsymbol{w}^i)$, where ∇ denotes the gradient operator and η is the *learning rate* whose task is to control the size of the weight updates. Finding the gradient $\nabla L_{\text{avg}}(\boldsymbol{w})$ is done using the *backpropagation* algorithm (Hecht-Nielsen 1992).

Usually, the training data is split into *batches* that are much smaller than the whole data set; this speeds up the process considerably since the batches usually approximate the gradient of the average loss well. One step of batch gradient descent is referred to as an *iteration*, and a pass over the whole data set is referred to as an *epoch*. A popular algorithm is *stochastic gradient descent* (Bottou 2010) where each batch is populated with training examples at random without replacement.

2.2.2 Avoiding Overfitting

Dropout (Srivastava et al. 2014) is a very powerful technique to avoid training data bias (overfitting). The idea is that a fraction ρ of the neurons in a particular layer are inactivated at training time by setting them to 0. This technique has proven to provide major improvements compared to other regularization techniques.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) apply convolution, non-linear activation (such as ReLU) and pooling in sequence in an attempt to extract meaningful features from spatial data such as images. Each of these operations can be regarded as a distinct layer of the CNN that transforms the data upstream. The fundamental building blocks of the CNN, convolution and pooling layers, are presented here. For a thorough walkthrough of CNNs, consult *the book* by Goodfellow, Bengio, and Courville (2016).

2.3.1 The Convolution Layer

Convolution in CNNs is done by sliding a *filter* or *kernel* across an input, e.g. a pixel map of a black-and-white image as illustrated in fig. 2.4. Imagine that the filter in fig. 2.5 is placed on top of the image so that the top-left corner of the filter is on top of the top-left corner of the image. The numbers that lie on top of one another is simply multiplied together, and then summed to produce a single value. The filter is then shifted by the *stride length*, and the calculation is re-done for a different area of the image. This shifting is repeated until the entire top row of the image has been convolved. When the entire top row of the image has been convolved, the kernel is instead shifted downwards by the stride length, and the next row is convolved in the same fashion. Figure 2.6 shows the result of applying the kernel in fig. 2.5 with a stride length of 1 to the input in fig. 2.4.

2.3.2 The Pooling Layer

Pooling is usually done after convolution and non-linear activation (such as applying ReLU to the output of a convolution) to reduce dimensionality. A popular type of pooling is *max pooling*, where a fixed-size portion of the input is in focus, and the maximum element of this portion is the representative.

1	1	0
1	0	0
0	1	1

Figure 2.4: Sample input to convolution layer representing a 3×3 black-and-white image.

0	1
1	0

Figure 2.5: Sample 2×2 convolution filter.

2	0
0	1

Figure 2.6: Result of applying the filter in fig. 2.5 with a stride of 1 to the input in fig. 2.4.

Consider the input in shown in fig. 2.7. We want to apply max pooling with a 2×2 window and a stride length of two. We first consider the four elements in the top-left corner. To find the output of the pooling map corresponding to this area of the input, we simply compute $\max(9, -3, 5, 3) = 9$. We then slide the pooling window two elements to the right to find the next output, which is $\max(-5, 8, 1, 16) = 16$. The window behaves in the same manner as the filter in a convolution, but instead of summing element-wise products, we report the maximum element. The result of performing max pooling with a 2×2 window and a stride length of two on the input in fig. 2.7 is shown in fig. 2.8.

9	-3	-5	8
5	3	1	16
7	6	3	-4
-2	1	1	8

Figure 2.7: Sample 4×4 input to a max pooling layer.

9	16
7	8

Figure 2.8: Result of performing max pooling to the input in fig. 2.7.

Chapter 3

Bayesian Optimization

Bayesian Optimization (BO) is a sequential design strategy for global minimization of noisy and expensive black-box functions f. Since f is noisy, it cannot be evaluated directly—a point x can be queried to obtain $f(x) \sim m + \epsilon$ where m is the mean function value at x and ϵ is a noise term drawn from some distribution that may depend on x.

The BO strategy has two constituent parts: the *surrogate* and the *acquisition function*. The surrogate serves to mimic f by modeling $p(f \mid x)$, i.e. the posterior of the function value given x, and is usually orders of magnitude cheaper to evaluate than querying f itself. The acquisition function is a heuristic that utilizes the surrogate to determine which point x to query f next.

Any BO design could be described as an iterative procedure where in iteration t the evaluation history of f, $\{(x_1, y_1), ..., (x_{t-1}, y_{t-1})\}$, is used to update the surrogate. The updated surrogate is then used by the acquisition function to determine which point to query f next. The general procedure is formalized in algorithm 1.

The goal of this section is to explain one of the most widely used surrogates, the Gaussian process; to present some of the most popular acquisition functions associated with this surrogate; and to demonstrate how Gaussian processes can be used to optimize hyperparameters of a machine learning model. As such, this section is structured as follows: section 3.1 deals with Gaussian processes in detail, section 3.2 deals with associated acquisition functions, and section 3.3 demonstrates how to use Gaussian processes to optimize the parameters of an SVM.

Algorithm 1 General BO strategy.

Input: Function f to optimize

$D \leftarrow \emptyset$
while within computational budget **do**
    Update surrogate model using D
    Obtain point x^* that maximizes the acquisition function
    Query f at x^* to obtain y^*
    $D \leftarrow D \cup (x^*, y^*)$
end while

3.1 Gaussian Processes

The formulae and derivations presented in section 3.1 are borrowed from Rasmussen (2003) and Rasmussen and Williams (2006).

Multivariate Gaussian *distributions* are distributions over vectors. In a standard multivariate Gaussian distribution, the stochastic variable of interest $\boldsymbol{X}$ is a k-dimensional vector, and $1 \leq i \leq k$ indexes the ith variable of X. The distribution of $\boldsymbol{X}$ can be fully specified by a mean vector $\boldsymbol{\mu}$ and a covariance matrix Σ.

Gaussian *processes* (GPs), on the other hand, are distributions over *functions*. Whereas a multivariate Gaussian distribution is specified by a mean vector $\boldsymbol{\mu}$ and covariance matrix Σ, a GP is fully specified by its mean function $m(\boldsymbol{x})$ and covariance (or kernel) function $k(\boldsymbol{x}, \boldsymbol{x}')$, and if f is distributed as a GP with mean function m and covariance function k, we write $f \sim \mathcal{GP}(m, k)$. Another difference of the GP is the way it is indexed. In a multivariate Gaussian distribution, the positions in the stochastic vector index the individual variables. In a GP, it is the argument $\boldsymbol{x}$ that plays the role of index—for every argument $\boldsymbol{x}$, there is a corresponding stochastic variable $f(\boldsymbol{x})$.

Let's take a look at a practical example. Suppose we are given the mean function $m(x) = 0$ and the covariance function $k(\boldsymbol{x}, \boldsymbol{x}') = \exp\big(-\frac{|\boldsymbol{x}-\boldsymbol{x}'|^2}{2}\big)$, and we wish to sample function values for $\boldsymbol{x} \in \{2, 4, 6\}$. The mean vector is then $\boldsymbol{\mu} = [0, 0, 0]$, and the covariance matrix is a 3×3 matrix where entry (i, j) is $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Note that the choice of order of the $\boldsymbol{x}$-values does not matter. If we order the $\boldsymbol{x}$-values by magnitude, the entry in the second row and third column is $k(4, 6) = \exp\big(-\frac{|4-6|^2}{2}\big) = \exp\big(-2\big)$, and the entire covariance matrix is

$$\begin{bmatrix} k(2,2) & k(2,4) & k(2,6) \\ k(4,2) & k(4,4) & k(4,6) \\ k(6,2) & k(6,4) & k(6,6) \end{bmatrix} = \begin{bmatrix} 0 & \exp\big(-2\big) & \exp\big(-8\big) \\ \exp\big(-2\big) & 0 & \exp\big(-2\big) \\ \exp\big(-8\big) & \exp\big(-2\big) & 0 \end{bmatrix}. \quad (3.1)$$

The mean vector and covariance matrix specify a standard multivariate Gaussian distribution. Consequently, the function values at the points specified can be sampled using standard sampling techniques.

3.1.1 Updating the Prior

If we obtain information about the stochastic function f in terms of a *training* vector of evaluations $\boldsymbol{f}$ at known points $\boldsymbol{x}_i$, we can update the mean and covariance functions to reflect the acquired information.

Imagine now that we're interested in calculating the posterior distribution of some *test* evaluations $\boldsymbol{f}_*$ at inputs $\boldsymbol{x}_j$ given training function evaluations $\boldsymbol{f}$ at inputs $\boldsymbol{x}_i$, i.e. we're interested in calculating $\boldsymbol{f}_*|\boldsymbol{f}$.

Recall that if $\boldsymbol{x}$ and $\boldsymbol{y}$ follow a joint Gaussian distribution, i.e.

$$\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} \sim \mathcal{N}\bigg(\begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{bmatrix}, \begin{bmatrix} A & C \\ C^{\mathrm{T}} & B \end{bmatrix} \bigg), \quad (3.2)$$

then the posterior $\boldsymbol{x}|\boldsymbol{y}$ can be calculated as

$$\boldsymbol{x}|\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{a} + CB^{-1}(\boldsymbol{y} - \boldsymbol{b}), A - CB^{-1}C^{\mathrm{T}}). \quad (3.3)$$

Furthermore, since by design

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f}_* \end{bmatrix} \sim \mathcal{N}\bigg(\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{bmatrix}, \begin{bmatrix} \Sigma & \Sigma_* \\ \Sigma_*^{\mathrm{T}} & \Sigma_{**} \end{bmatrix} \bigg), \quad (3.4)$$

where Σ denotes the training set covariances, Σ_* the training-test set covariances and Σ_{**} the test set covariances, it follows directly from equation 3.3 that

$$\boldsymbol{f}_*|\boldsymbol{f} \sim \mathcal{N}(\boldsymbol{\mu}_* + \Sigma_*^{\mathrm{T}}\Sigma^{-1}(\boldsymbol{f} - \boldsymbol{\mu}), \Sigma_{**} - \Sigma_*^{\mathrm{T}}\Sigma^{-1}\Sigma_*). \quad (3.5)$$

From equation 3.5 it follows that the mean and covariance functions of the Gaussian process f conditioned on observational data $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y})$ where $\boldsymbol{X}$ is

a vector of training inputs and $\boldsymbol{y}$ a vector of corresponding training function values is given by

$$\begin{aligned} f|\mathcal{D} &\sim \mathcal{GP}(m_{\mathcal{D}}, k_{\mathcal{D}}), \\ m_{\mathcal{D}}(\boldsymbol{x}) &= m(\boldsymbol{x}) + \Sigma(\boldsymbol{X}, \boldsymbol{x})^{\mathrm{T}} \Sigma^{-1} (\boldsymbol{y} - \boldsymbol{\mu}) \\ k_{\mathcal{D}}(\boldsymbol{x}, \boldsymbol{x}') &= k(\boldsymbol{x}, \boldsymbol{x}') - \Sigma(\boldsymbol{X}, \boldsymbol{x})^{\mathrm{T}} \Sigma^{-1} \Sigma(\boldsymbol{X}, \boldsymbol{x}'). \end{aligned} \tag{3.6}$$

Here, $\Sigma(\boldsymbol{X}, \boldsymbol{x})$ is a vector of covariances between every training case and $\boldsymbol{x}$, Σ is the covariance matrix for $\boldsymbol{X}$ and $\boldsymbol{\mu} = [m(\boldsymbol{x}_1), ..., m(\boldsymbol{x}_n)]^{\mathrm{T}}$.

3.1.2 Kernel Functions

All the kernels described here can be found in Rasmussen and Williams (2006).

The simplest kernel is the *constant* kernel, which is simply defined as

$$k_{\mathrm{C}}(\boldsymbol{x}, \boldsymbol{x}') = C. \tag{3.7}$$

Sometimes the stochastic function f may have additional variance with itself, in which case the *white Gaussian noise* kernel can be used, which is defined as

$$k_{\mathrm{GN}}(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2 \delta_{i,i'}, \tag{3.8}$$

where $\delta_{i,i'}$ is the Kronecker-delta, which is related to the inputs $\boldsymbol{x}_i$. It is equal to 1 when $i = i'$ for the inputs $\boldsymbol{x}_i$, and 0 otherwise. This means that although $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ may be different cases with the same value for the input, the covariance between the cases will be 0.

A popular choice of kernel is the *squared exponential* kernel, which is defined as

$$k_{\mathrm{SE}}(\boldsymbol{x}, \boldsymbol{x}') = exp\bigg(- \frac{d^2}{2l^2} \bigg). \tag{3.9}$$

Here, $d = |\boldsymbol{x} - \boldsymbol{x}'|$, and l is the *length scale*. The length scale controls how close points have to be to influence each other significantly. As l increases, the value of the kernel approaches 1, and as l decreases, the value of the kernel approaches 0. In other words, the larger the length scale, the further away points can be while still influencing each other.

$$k_{\mathrm{SE}}(\boldsymbol{x}, \boldsymbol{x}') = exp\bigg(- \frac{r^2}{2} \bigg) \tag{3.10}$$

where $r = \sqrt{\sum_{i=1}^{n} \left(\frac{\boldsymbol{x}_i - \boldsymbol{x}'_i}{l_i} \right)^2}$.

The most widely used kernel is perhaps the Matérn kernel, which is defined as

$$k_{\text{Matern}}(\boldsymbol{x}, \boldsymbol{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)}\Big(\frac{\sqrt{2\nu}d}{l}\Big)^{\nu} K_{\nu}.\Big(\frac{\sqrt{2\nu}d}{l}\Big) \tag{3.11}$$

Here, K_ν is the modified Bessel function, d and l are defined identically as for the exponential kernel, and ν is a user-specified parameter which controls the smoothness of the functions sampled. In Rasmussen and Williams (2006) it is argued that perhaps the best values for ν for machine learning are $\frac{3}{2}$ and $\frac{5}{2}$, because smaller or larger values impose functions that are either too rough or too smooth. For these values of ν, the kernel expressions simplify to

$$k_{\nu=3/2}(d) = \left(1 + \frac{\sqrt{3}d}{l}\right) exp\left(-\frac{\sqrt{3}d}{l}\right), \tag{3.12}$$

and

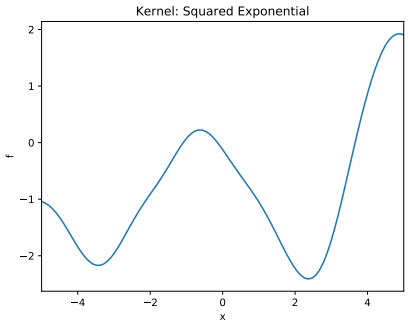
$$k_{\nu=5/2}(d) = \left(1 + \frac{\sqrt{5}d}{l} + \frac{5d^2}{3l^2}\right) exp\left(-\frac{\sqrt{5}d}{l}\right). \tag{3.13}$$

For the squared exponential and the Matérn kernel, it possible to let each individual dimension of $\boldsymbol{x}$ have its own length scale, in which case every occurence of $\frac{d}{l}$ is replaced with $r = \sqrt{\sum_{i=1}^{n} \big(\frac{\boldsymbol{x}_i - \boldsymbol{x}_i'}{l_i}\big)^2}$.

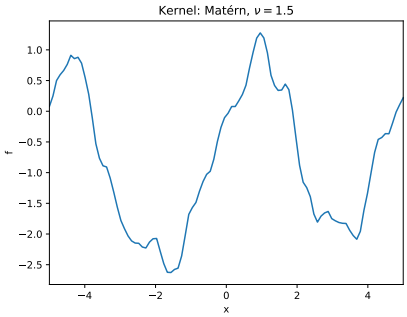
The squared exponential kernel and the Matérn kernel, or compositions built on them, are the most frequently used kernels. The samples generated from GPs using these kernels are very different on a qualitative level, however. This is illustrated in fig. 3.1. The plots in fig. 3.1a, fig. 3.1b and fig. 3.1c show GP samples generated at inputs $x \in [-5, 5]$ for different kernels and zero mean. Note the difference in smoothness of the curves connecting the samples. The curve in fig. 3.1a, corresponding to the Squared Exponential kernel, is very smooth, while the curve in fig. 3.1b, corresponding to the Matérn kernel with $\nu = 1.5$, is very rough. The curve in fig. 3.1c, corresponding to the Matérn kernel with $\nu = 2.5$, falls somewhere inbetween the two others in terms of smoothness. Note that larger values of ν for the Matérn kernel correspond to smoother curves. Thus, if we have some knowledge about the smoothness of a function we're trying to model with a GP, we can incorporate this knowledge by choosing the kernel we think mimics the smoothness of the function.

3.1.3 A Quick Note on Non-stationarity

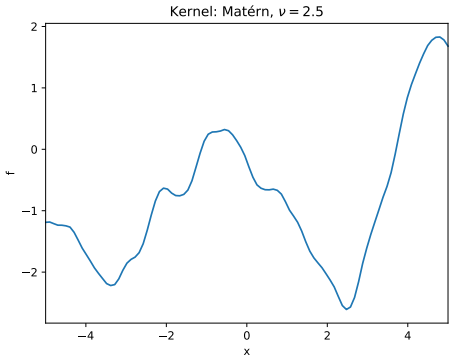
All the kernels presented here are *non-stationary*—they are invariant to translations in the input space, i.e. $k(\boldsymbol{x} + \boldsymbol{r}, \boldsymbol{x}' + \boldsymbol{r}) = k(\boldsymbol{x}, \boldsymbol{x}')$ for arbitrary $\boldsymbol{r}$. This



(a) Samples generated from GP with the Squared Exponential kernel and zero mean. The curve connecting the samples appears smooth.



(b) Samples generated from GP with the Matérn kernel ($\nu = 1.5$) and zero mean. The curve connecting the samples appears jagged.



(c) Samples generated from GP with the Matérn kernel ($\nu = 2.5$) and zero mean. The curve connecting the samples appears less jagged than for the Matérn kernel with $\nu = 1.5$, but not as smooth as for the Squared Exponential kernel.

Figure 3.1: Samples generated at different values of x from GPs using different kernels. The length scale l was set to 1 for each kernel, and the mean function was 0 for all GPs.

means that functions that vary significantly across length scales—e.g. a function that peaks sharply around some points and is relatively smooth elsewhere—are difficult to model with a GP with such kernels. One solution to this problem is to model the function in a transformed input space using an appropriate bijective transformation. Finding an appropriate transformation is difficult however, but a popular choice is the logarithmic transformation, where the desired dimensions of each input $\boldsymbol{x}$ to the GP are transformed to their respective logarithms. In Snoek, Swersky, et al. (2014), the authors propose a methodology for automatically learning a wide family of transformations using the cumulative Beta distribution function. Specifically, they alter the kernel function to be $k(w(\boldsymbol{x}), w(\boldsymbol{x}'))$, where

$$w_d(\boldsymbol{x}_d) = \int_0^{\boldsymbol{x}_d} \frac{u^{\alpha_d - 1}(1-u)^{\beta_d - 1}}{B(\alpha_d, \beta_d)} \mathrm{d}u. \tag{3.14}$$

In eq. (3.14), α_d and β_d are free parameters, and $B(\alpha_d, \beta_d)$ is a normalization constant. Their approach sees large improvements compared to methods that do not transform the input space of the GP, but the benefit of their method compared to a simple prior transformation (such as the log transformation) remains unclear. .

3.1.4 Fitting Hyperparameters

You may have noted that GPs have hyperparameters of their own, defined by the choice of kernel and mean function. How do we choose the hyperparameters θ of the GP that best explain observational data? The simplest way to fit the hyperparameters θ of a GP to observational data $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y})$ is to maximize $L = \log p(\boldsymbol{y}|\boldsymbol{X}, \theta)$, which is the logarithm of the likelihood. Since the observational data is Gaussian, L has an analytic expression (Rasmussen 2003), namely

$$L = -\frac{1}{2}\log|\Sigma| - \frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu})^{\mathrm{T}}\Sigma^{-1}(\boldsymbol{y} - \boldsymbol{\mu}) - \frac{n}{2}\log(2\pi), \tag{3.15}$$

where Σ is a covariance matrix derived from applying the kernel to $\boldsymbol{X}$, $\boldsymbol{\mu}$ are the predicted means for the inputs, and $n = \dim(\boldsymbol{x})$. This expression can then be optimized using methods such as gradient descent.

3.2 Common Acquisition Functions

The goal of the acquisition function is to balance exploration versus exploitation in the hyperparameter search. Regions of the hyperparameter space that seem

promising should be prioritized over regions that seem less promising, but there shouldn't be large, unexplored regions either.

The formulae of the acquisition functions presented in section 3.2 concern GP surrogates. As notational convention, we let $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ denote the predicted mean and standard deviation of an arbitrary GP, respectively.

3.2.1 Probability of Improvement

Probability of Improvement (PI) (Kushner 1964) computes the probability that the input evaluated is an improvement over the current best input, and it can be expressed as

$$\alpha_{PI}(\boldsymbol{x}) = \Phi(\gamma(\boldsymbol{x})), \qquad \gamma(\boldsymbol{x}) = \frac{f(\boldsymbol{x}_{best}) - \mu_n(\boldsymbol{x})}{\sigma_n(\boldsymbol{x})}, \tag{3.16}$$

where $\Phi(\cdot)$ is the cumulative density function of the standard normal distribution. PI is often criticized for being overly exploitative.

3.2.2 Expected Improvement

Expected Improvement (EI) (Mockus, Tiesis, and Zilinskas 1978) computes the expected improvement of the input $\boldsymbol{x}$ with respect to the current best input $\boldsymbol{x}_{best}$. It can be expressed as

$$\alpha_{EI}(\boldsymbol{x}) = \sigma_n(\boldsymbol{x})(\gamma(\boldsymbol{x})\Phi(\gamma(\boldsymbol{x})) + \mathcal{N}(\gamma(\boldsymbol{x}); 0, 1)), \tag{3.17}$$

where $\mathcal{N}(\gamma(\boldsymbol{x}); 0, 1))$ is the standard normal density function evaluated at $\gamma(\boldsymbol{x})$. EI is regarded as well-balanced and robust when it comes to exploitation versus exploration.

3.3 A Practical Example: Optimizing a Single SVM Hyperparameter

To demonstrate how BO can be used to optimize the hyperparameters of a machine learning model, we will be optimizing a single hyperparameter for a one-vs-one SVM, γ, as defined in section 2.1.3 on the `digits` dataset from Scikit-learn (Pedregosa et al. 2011). We'll be using the Python libraries `Scikit-learn` (Pedregosa et al. 2011), `SciPy` (Virtanen et al. 2020) and `NumPy` (Oliphant 2006).

The acquisition function we'll be using is EI (eq. (3.17)), which in Python can be written as:

```
from scipy.stats import norm

def ei(gp, incumbent, x):
    mean, std = gp.predict([x], return_std=True)
    gamma = (incumbent - mean[0])/std[0]
    f = std[0]*(gamma*norm.cdf(gamma) + norm.pdf(gamma))
    return f
```

The function `ei` has three arguments: `gp`, an object of the class `sklearn.gaussian_process.GaussianProcessRegressor`; `incumbent`, a scalar value that corresponds to $f(\boldsymbol{x}_{best})$ in eq. (3.17); and `x`, a variable that corresponds to the hyperparameters we are trying to optimize, which in our case is just γ.

Since this is a BO procedure, we want to locate the point that maximizes the EI in every iteration. Thus, the EI has to be optimized. This can be accomplished using `scipy.optimize.dual_annealing` (or any other optimizer, the choice here is arbitrary):

```
from scipy.optimize import dual_annealing

def ei_opt(gp, incumbent, bounds):
    return dual_annealing(lambda x: -ei(gp, incumbent, x),
    bounds).x
```

Note that since `dual_annealing` is a *minimizer*, we have to optimize the EI. The arguments `gp` and `incumbent` are defined as they are for `ei`. The argument `bounds` is a list of tuples that indicate valid ranges for each dimension of `x`. We'll be setting the `bounds` for γ to $[(10^{-5}, 10^{5})]$, which in log space is $[-5, 5]$.

Now to the actual optimization. We want to minimize the test error rate of our SVM classifier using BO. The kernel we'll be using is $k_{\nu=5/2} + k_{\text{GN}}$. Additionally, since γ is allowed to vary over several orders of magnitudde, we'll be modeling γ in log space for the GP:

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, WhiteKernel
from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)
n_iter = 20
param_hist = []
error_hist = []
min_error_hist = []
bounds = [(-5, 5)]
```

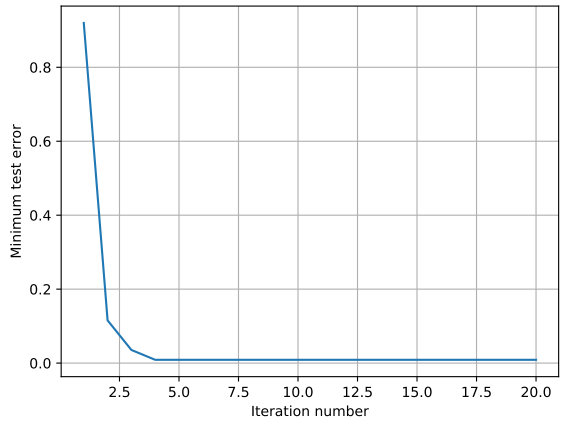
```
incumbent = 1
# can't get worse than 100% error rate
for i in range(n_iter):
    kernel = Matern(nu=2.5) + WhiteKernel()
    gp = GaussianProcessRegressor(kernel=kernel,
    normalize_y=True)
    if param_hist:
        gp.fit(param_hist, error_hist)
    # since GP models in log space, remember to transform back
    gamma = 10**ei_opt(gp, incumbent, bounds)[0]
    svm = SVC(gamma=gamma)
    svm.fit(X_train, y_train)
    error = 1 - svm.score(X_test, y_test)
    incumbent = min(incumbent, error)
    # we're modeling the parameter in log space
    param_hist.append([np.log10(gamma)])
    error_hist.append(error)
    min_error_hist.append(incumbent)
```

In each iteration of the BO procedure above, the incumbent is used along with a GP fit on the previous $(\gamma, \text{test error})$-pairs to find the point that maximizes the EI. The code follows the procedure outlined in algorithm 1. We can plot the minimum test error as a function of the iterations and EI as a function of $\log(\gamma)$ using `Matplotlib` (Hunter 2007):

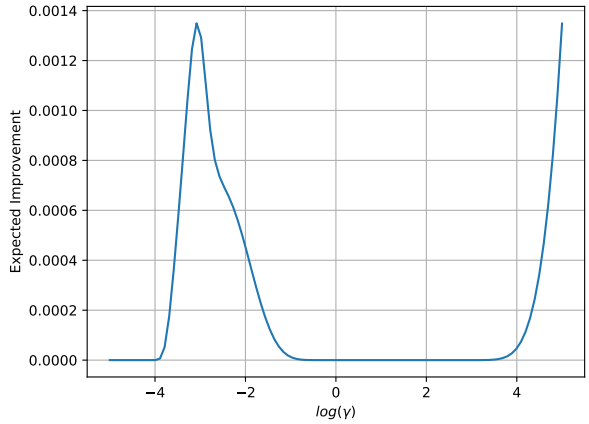
```
import matplotlib.pyplot as plt
import os

savedir = "Illustrations"
plt.plot(range(1, n_iter+1), min_error_hist)
plt.xlabel("Iteration number")
plt.ylabel("Minimum test error")
plt.grid(b=True)
plt.savefig(os.path.join(savedir, "error-vs-iteration.pdf"))

plt.figure()
gammas = np.linspace(-5, 5, 100)
ei_of_gammas = [ei(gp, incumbent, [gamma]) for gamma in gammas]
plt.plot(gammas, ei_of_gammas)
plt.xlabel(r"$log(\gamma)$")
plt.ylabel("Expected Improvement")
plt.grid(b=True)
plt.savefig(os.path.join(savedir, "expected_improvement.pdf"))
```

The results from this particular run can be seen in fig. 3.2a and fig. 3.2b. We see that the method quickly converges to an error rate that is very close to 0, and that the expected improvement is very small in magnitude, which means that the expected gains are small. That makes sense, since an error rate below

0 is impossible.



(a) The minimum error as a function of the number of iterations.



(b) EI as a function of $\log(\gamma)$.

Figure 3.2: The plots obtained from running the code outlined in section 3.3.

Chapter 4

Hyperband

Hyperband (HB) (Li et al. 2017) is an algorithm for hyperparameter optimization, and has been shown to be capable of providing orders-of-magnitude speedups over competing methods such as random search or BO. The algorithm works by means of *adaptive resource allocation*. What that means, essentially, is that configurations that are observed to perform poorly relative to other configurations are discarded. Configurations that perform well are allocated more resources. This cycle of discarding and allocating more resources is repeated until the pre-defined maximum possible resource is allocated to each configuration. Thus, the way Hyperband approaches hyperparameter optimization is essentially intelligently allocating resources so that the number of possible configurations to test is maximized while still preserving a preferential structure between the individual configurations. Hyperband is based on another algorithm which is called SuccessiveHalving; this algorithm will be presented next.

4.1 SuccessiveHalving

The idea behind SuccessiveHalving (Jamieson and Talwalkar 2016) is pretty simple. Given n hyperparameter configurations and a resource budget B, allocate each of the n configurations $\frac{B}{n}$ resources each, train them, compute each configuration's performance on the test data, and discard the worst half as dictated by the performance metric. Repeat this process, each time doubling the resource allocated to each configuration until one configuration remains as the winner.

The problem with SuccessiveHalving is that n is an input to the algorithm. With a pre-defined finite budget B (such as the number of epochs to train neural networks for a given batch size), $\frac{B}{n}$ resources are allocated to each configuration on average (Li et al. 2017). It is not clear beforehand whether a large ratio of $\frac{B}{n}$, i.e. fewer configurations with a larger budget per configuration, should be preferred over a small ratio, i.e. more configurations with a smaller budget per configuration. A smaller ratio would allow for more configurations to be tested, but with the potential drawback of it being possible that good configurations are hard to differentiate from bad configurations since sufficient resources had not been allocated, consequently ending up with a worse final configuration than if we had chosen a smaller ratio for $\frac{B}{n}$. Hyperband proposes a solution to this problem.

4.2 Description of Hyperband

Hyperband, shown in algorithm 2, takes two inputs: R, the maximum resource that can be allocated to any configuration, and η, a discarding factor associated with SuccessiveHalving. Hyperband approaches the "$\frac{B}{n}$ ratio problem" through considering multiple values of n for a fixed budget B. For each value of n, there is an associated value r that dictates the minimum resource that can be allocated to a single configuration. Hyperband essentially performs a geometric search through possible values of n, each time increasing n by approximately a factor η.

For each fixed (n, r)-pair, SuccessiveHalving is performed for n hyperparameter configurations sampled at random, but instead of keeping the best half of the configurations, $\frac{1}{\eta}$ are kept, and the resource to allocate to each configuration is increased by a factor η instead of 2. Each such round of SuccessiveHalving is also referred to as a bracket, and is designed to use approximately B resources. Thus, Hyperband is a solution to the "$\frac{B}{n}$ ratio problem" at only $s_{max} + 1$ times the cost that SuccessiveHalving is.

Algorithm 2 The Hyperband algorithm.

Input: R, η

$s_{max} \leftarrow \lfloor \log_\eta(R) \rfloor$
$B \leftarrow (s_{max} + 1)R$
for $s \in \{s_{max}, s_{max} - 1, ..., 0\}$ **do**
    $n \leftarrow \lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil$
    $r \leftarrow R\eta^{-s}$
    $T \leftarrow$ get_random_configurations(n)
    for $i \in \{0, 1, ..., s\}$ **do**
        $n_i \leftarrow \lfloor n\eta^{-i} \rfloor$
        $r_i \leftarrow r\eta^i$
        $L \leftarrow \{\text{run\_then\_return\_val\_loss}(x, r_i) : x \in X\}$
        $T \leftarrow \text{top\_k}(T, L, \lfloor n_i/\eta \rfloor)$
    end for
end for
return hyperparameter configuration with smallest intermediate loss seen so far

Hyperband invokes a couple of subroutines, and they are:

- get_random_configurations(n): A method that returns n i.i.d. hyperparameter configurations from any underlying probability distribution.
- run_then_return_val_loss(x, r): A method that trains configuration x on r resources and returns the validation loss. Note that it doesn't necessarily have to be the validation loss that is returned. Any scoring metric where a lower value is preferred over a larger one can be used, such as the validation error.
- top_k(T, L, k): A method that takes as input a list of configurations T and their validation losses L and returns the k best-performing configurations.

4.3 Setting the Parameters

The maximum resource R influences the number of brackets in Hyperband, since the number of brackets is $s_{max} + 1$ and $s_{max} = \log_\eta(R)$. The value of R also influences the number of configurations n in the most aggressive bracket of Hyperband, which is when $s = s_{max}$. The larger R is, the larger n will

be. Sometimes, we may want to limit the number of configurations to n_{max} in the most aggressive bracket of Hyperband to avoid overhead associated with training and testing models. In that case, the alteration of the algorithm is very simple. Simply set $s_{max} = \lfloor \log_\eta(n_{max}) \rfloor$, and everything else remains the same as before.

The parameter η controls the proportion of configurations to discard in every iteration of `SuccessiveHalving`. According to Li et al. (2017), the choice of η does not matter all that much, but Euler's number $e = 2.718...$ is shown to be theoretically optimal. The authors suggest leaving η at either 3 or 4.

4.4 Choosing Probability Distributions

In practice, any probability distribution can be used to sample hyperparameter configurations in Hyperband. Not all probability distributions are equal, though. For instance, when sampling a hyperparameter that may range over many orders of magnitude, it may make more sense to sample the parameter from a distribution where the logarithm of the parameter is uniformly distributed, rather than the parameter itself being uniformly distributed. This would ensure that on average, when sampling that parameter, every order of magnitude is represented in equal number, instead of smaller orders of magnitude being dominated by larger ones.

One point where Hyperband could be improved is by dynamically updating the probability distribution configurations are drawn from. Regions of the parameter space that perform poorly should have lower density than regions with promising parameters. On the other hand, regions that are unexplored should not lose density.

Chapter 5

Combining Approaches

5.1 Previous Work

Combining Hyperband and Bayesian optimization has been attempted in the past. This section summarizes, to the best of my knowledge, everything that has been done in the field on this particular matter.

Bertrand et al. (2017) propose a hybrid approach that uses Bayesian optimization for model selection. The configuration selection in the first bracket of their method is done with a uniform prior, but all subsequent selections are done with Bayesian optimization utilizing a surrogate trained on all evaluations performed so far. Their surrogate is a Gaussian process with the squared exponential kernel. They use the expected improvement acquisition function for configuration selection, but with a novel twist. Bayesian optimization is sequential in nature, so instead they normalize the expected improvement values to produce a probability distribution. They then sample the next point to evaluate from this probability distribution.

Unfortunately, no statistically significant conclusions can be drawn from their study, as their experiment was ran only once. They also made some questionable choices in my opinion, such as using the squared exponential kernel instead of the Matérn kernel. Furthermore, the article in question glosses over many details, and it's hard to extract exactly how their method is implemented.

J. Wang, Xu, and X. Wang (2018) propose a hybrid approach outlined in algorithm 3. In algorithm 3, α is the acquisition function. As can be observed, random search is simply replaced with Bayesian optimization. As Bayesian sur-

rogate they use the Tree-structured Parzen Estimator proposed by J. S. Bergstra et al. (2011). They observe that their combined approach can perform better than both Hyperband and Bayesian optimization on their own, especially on problems that are complex. Their approach is simple to understand and implement, but one weakness of their method is that they don't utilize all the data available from previous experiments for every value of s. For every value of s, the Bayesian optimization procedure is restarted from ground zero. Intuitively, one should use all the information that is available, so this must be thought of as a weakness of their method.

Algorithm 3 Method proposed by J. Wang, Xu, and X. Wang (2018).

Input: R, η
$s_{max} \leftarrow \lfloor \log_\eta(R) \rfloor$
$B \leftarrow (s_{max} + 1)R$
for $s \in \{s_{max}, s_{max} - 1, ..., 0\}$ **do**
  $n \leftarrow \lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil$
  $r \leftarrow R\eta^{-s}$
  for $i \in \{0, 1, ..., s\}$ **do**
    $n_i \leftarrow \lfloor n\eta^{-i} \rfloor$
    $r_i \leftarrow r\eta^i$
    if $i = 0$ **then**
      $X \leftarrow \emptyset$
      $D_0 \leftarrow \emptyset$
      for $t \in \{1, 2, ..., n_i\}$ **do**
        $x_{t+1} \leftarrow \text{argmax}_x \alpha(x|D_t)$
        $f(x_{t+1}) \leftarrow \text{run\_then\_return\_val\_loss}(x, r_i)$
        $X \leftarrow X \cup \{x_{t+1}\}$
        $D_{t+1} \leftarrow D_t \cup \{(x_{t+1}, f(x_{t+1}))\}$
        Update probabilistic surrogate model using D_{t+1}
      end for
    else
      $F \leftarrow \{\text{run\_then\_return\_val\_loss}(x, r_i) : x \in X\}$
      $X \leftarrow \text{top\_k}(X, F, \lfloor n_i/\eta \rfloor)$
    end if
  end for
end for
return hyperparameter configuration with lowest loss

Falkner, Klein, and Hutter (2018) propose a hybrid approach that replaces the random samples of regular Hyperband with the approach outlined in algorithm 4. The algorithm uses the observational data $D = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$, i.e. previous configuration-score pairs, to construct a distribution from kernel density estimators (KDEs) that can be queried for subsequent configurations. The algorithm relies on eq. (5.1) and eq. (5.2). In eq. (5.1), $\alpha = \min(y_1, ..., y_n)$. In eq. (5.2), $N_{b,l}$ and $N_{b,g}$ are the number of best and worst configurations respectively used to model $l(\boldsymbol{x})$ and $g(\boldsymbol{x})$, respectively.

Algorithm 4 refers to a quantity $l'(\boldsymbol{x})$, which is simply the same KDE as for $l(\boldsymbol{x})$, where each bandwidth is multiplied with a factor b_w to encourage exploration near promising configurations.

Their method performs strongly on several popular benchmarks compared to regular Hyperband, and seems to be insensitive to the choice of hyperparameters. A weakness of their method in my opinion is the fact that they ignore the data collected with a budget that is less than the current budget considered.

$$\begin{aligned} l(\boldsymbol{x}) &= p(y < \alpha | \boldsymbol{x}, D) \\ g(\boldsymbol{x}) &= p(y > \alpha | \boldsymbol{x}, D) \end{aligned} \tag{5.1}$$

$$\begin{aligned} N_{b,l} &= \max(N_{min}, q \cdot N_b) \\ N_{b,g} &= \max(N_{min}, N_b - N_{b,l}) \end{aligned} \tag{5.2}$$

Algorithm 4 The sampling method as proposed by Falkner, Klein, and Hutter (2018).

Input: Observations D, fraction of random runs ρ, percentile q, number of samples N_s, minimum number of points N_{min} to build a model (default $N_{min} = \#\text{hyperparameters} + 1$), and bandwith factor b_w
Output: Next configuration to evaluate

```
if rand(0, 1) < ρ then
    return random configuration
end if
b ← arg max{D_b : |D_b| ≥ N_min + 2}
if b = ∅ then
    return random configuration
end if
fit KDEs according to eq. (5.1) and eq. (5.2)
draw N_s samples according to l'(x) (see text)
return sample that maximizes l(x)/g(x)
```

5.2 Proposed Method

The general proposed method is reminiscent of the one presented by Bertrand et al. (2017), but with minor implementational differences. These differences will be explicitly pointed out. Note that Bertrand et al. (2017) used a GP as surrogate—both Falkner, Klein, and Hutter (2018) and J. Wang, Xu, and X. Wang (2018) utilized variations of TPE as presented by J. S. Bergstra et al. (2011).

The pseudo code for the general proposed method can be seen in algorithm 5. The idea is to update the sampling distribution as experimental results become available, and to utilize all the data that is available. In the first bracket of the algorithm, N_{min} configurations are sampled at random, whereas Bertrand et al. (2017) sampled all the configurations at random in the first bracket. These N_{min} configurations are trained and scored. The remaining $n - N_{min}$ configurations are obtained using the method `get_bayesian_configurations`(X, y, n). The method `get_bayesian_configurations`(X, y, n) takes as input the previous configurations and their associated budgets X, their losses y and the number of configurations to sample n. It is imperative that the budget for each configuration is included in X, since the budget explains variability for similar configurations trained on different budgets. Internally, the method should fit a

GP—the premise of the thesis is, after all, to use a GP as surrogate—on (X, y), and use some pre-defined acquisition function to sample hyperparameter configurations. I think what Bertrand et al. (2017) did is a good idea, i.e. using the normalized acquisition function to sample configurations. Sampling configurations in this manner liberates the BO procedure from training and testing sequentially, promoting parallelism.

The reason for sampling N_{min} configurations at random in the first bracket is that BO tends to perform poorly in the beginning when there is little data—the decisions of the strategy will be rather arbitrary and uninformed. In that case, an informed prior is probably a better idea. However, when the first N_{min} configurations and their associated scores become available, the configurations proposed by the Bayesian method will hopefully be well-informed. I propose a default value of $d + 1$ for N_{min}, just as Falkner, Klein, and Hutter (2018) did, where d is the number of hyperparameters. If $d + 1 < R$, which it is in most cases, the Bayesian sampler will come sooner into effect than for the algorithm proposed by Bertrand et al. (2017), which hopefully leads to comparatively faster convergence.

Algorithm 5 Proposed method to combine Hyperband and Bayesian optimization.

Input: R, η, N_{min}

$s_{max} \leftarrow \lfloor \log_\eta(R) \rfloor$
$B \leftarrow (s_{max} + 1)R$
$X \leftarrow \emptyset$
$y \leftarrow \emptyset$
for $s \in \{s_{max}, s_{max} - 1, ..., 0\}$ **do**
    $n \leftarrow \lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil$
    $r \leftarrow R\eta^{-s}$
    if $s = s_{max}$ **then**
        $T_r \leftarrow$ get_random_configurations(N_{min})
        $L_r \leftarrow \{\text{run\_then\_return\_val\_loss}(t_r, r) : t_r \in T_r\}$
        $X \leftarrow X \cup \{(t_r, r) : t_r \in T_r\}$
        $y \leftarrow y \cup L_r$
        $T \leftarrow$ get_bayesian_configurations($X, y, n - N_{min}$)
    else
        $T \leftarrow$ get_bayesian_configurations(X, y, n)
    end if
    for $i \in \{0, 1, ..., s\}$ **do**
        $n_i \leftarrow \lfloor n\eta^{-i} \rfloor$
        $r_i \leftarrow r\eta^i$
        $L \leftarrow \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$
        $X \leftarrow X \cup \{(t, r_i) : t \in T\}$
        $y \leftarrow y \cup L$
        if $s = s_{max}$ **and** $i = 0$ **then**
            $T \leftarrow T_r \cup T$
            $L \leftarrow L_r \cup L$
        end if
        $T \leftarrow \text{top\_k}(T, L, \lfloor n_i/\eta \rfloor)$
    end for
end for
return hyperparameter configuration with lowest loss

5.3 Implementational Details

The language used to implement the approach outlined in algorithm 5 was Python. The implementational details of some of the methods used will be presented and discussed here. Note that the code does not necessarily reflect algorithm 5 perfectly, but it follows the general idea.

The method `random_sample(config_spec)` takes as input a *configuration specification* that is assumed to be a dictionary, and it is the method that is responsible for generating random configurations in algorithm 5:

```
import random

def random_sample(config_spec):
    config = {}
    for param_spec in config_spec:
        lower_bound, upper_bound = param_spec["bounds"]
        pname = param_spec["name"]
        pscale = param_spec["scale"]; ptype = param_spec["type"]
        if ptype == "int" and pscale == "uniform":
            config[pname] = random.randint(lower_bound,
            upper_bound)
        elif ptype == "int" and pscale == "log":
            config[pname] = int(round(log_uniform(lower_bound,
            upper_bound)))
        elif ptype == "float" and pscale == "uniform":
            config[pname] = uniform(lower_bound, upper_bound)
        elif ptype == "float" and pscale == "log":
            config[pname] = log_uniform(lower_bound, upper_bound)
    return config


def uniform(lower, upper):
    return lower + (upper - lower)*random.random()



def log_uniform(lower, upper):
    return 10**uniform(lower, upper)
```

The dictionary should contain the keys `name`, `scale`, `type` and `bounds`. An example of a valid hyperparameter configuration is:

```
{
    "name": "gamma",
    "scale": "log",
    "type": "float"
    "bounds": (-10, 10),
}
```

The key `name` is associated with the name of the hyperparameter, `scale` is either `log` or `uniform` which dictates whether the parameter is drawn from a log-uniform or uniform distribution respectively, `type` tells the method whether the parameter is an integer or a floating point number, and `bounds` is a tuple to indicate the legal range of the parameter. The method returns a hyperparameter configuration that conforms to the specification. The method also utilizes helper methods `uniform()` and `log_uniform()` that returns a uniformly or log_uniformly distributed sample, respectively.

The method `bayesian_samples(X_gpr, y_gpr, log_R, bounds, n_samples, n_steps_mcmc)` is responsible for generating samples in a Bayesian way. It is defined as follows:

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import (Matern,
ConstantKernel, WhiteKernel)

def bayesian_samples(X_gpr, y_gpr, log_R, bounds,
n_samples, n_steps_mcmc=10**5):
    ls = [1 for _ in X_gpr[0]]
    kernel = Matern(nu=2.5, length_scale=ls) * ConstantKernel()
    kernel += WhiteKernel()
    gp = GaussianProcessRegressor(kernel=kernel, normalize_y=True)
    gp.fit(X_gpr, y_gpr)
    f = lambda x: ei(gp, min(y_gpr), np.append(x, log_R))
    return slice_sampler(n_steps_mcmc, f, bounds)[-n_samples:]
```

The method takes as input the previous hyperparameter configurations with their used budgets `X_gpr`, the configuration losses `y`, `log_R` which is the logarithm of the maximum budget, the bounds of the hyperparameters `bounds`, the number of samples `n_samples` and `n_steps_mcmc`, a parameter we'll come back to in a bit. The method fits a GP using the GP-related classes from Scikit-learn (Pedregosa et al. 2011) on `X_gpr` and `y`. The fitting is accomplished under the hood through the approach outlined in section 3.1.1 and section 3.1.4. The kernel of choice is the Matérn kernel with $\nu = 2.5$ times a constant as popularized by Snoek, Larochelle, and Adams (2012) added together with a white Gaussian noise kernel to account for auto-variance. This choice of kernel is in contrast to the squared exponential kernel used by Bertrand et al. (2017). This GP is used to construct a function that returns the expected improvement with the budget fixed at the maximum. I presume that the loss will exhibit non-stationarity in the budget dimension, and hence the budgets are assumed to be represented logarithmically for reasons described in section 3.1.3; the maximum budget is expected to be log-transformed as well. To my understanding, Bertrand et al. (2017) allowed the budgets to vary, but in my opinion, the budget should be

fixed; we're not interested in sampling configurations that have high probability because of uncertainty associated with the budget—we're controlling the budget in each bracket. It's fixed at the maximum to try to extrapolate to the final loss. The function that computes the expected improvement is then used as a probability distribution to sample probable configurations. To produce samples from this distribution, I opted to use slice sampling (Neal 2003) because it is simple to implement, and requires minimal tuning compared to other methods such as Metropolis-Hastings (Chib and Greenberg 1995). The parameter `n_steps_mcmc` controls how many such samples are produced. The reason we can't just use `n_samples` as the number of samples produced by slice sampling is that there is usually a burn-in period of such methods, where the samples have not yet migrated to a high-probability region. My Python implementation of multivariate slice sampling is provided below. The implementation utilizes NumPy (Oliphant 2006).

```
import numpy as np

def slice_sampler(n, f, bounds):
    def hyperrect_sample(hr):
        x = []
        for (l, r) in hr:
            x.append(np.random.uniform(l, r))
        return x

    # find first point where f > 0
    x = hyperrect_sample(bounds)
    while f(x) <= 0:
        x = hyperrect_sample(bounds)

    samples = [x]
    for i in range(n-1):
        hr = [[l, r] for (l, r) in bounds]
        # define the slice
        y = np.random.uniform(0, f(samples[-1]))
        x = hyperrect_sample(hr)
        while f(x) <= y:
            for i in range(len(x)):
                if x[i] < samples[-1][i]:
                    hr[i][0] = x[i]
                else:
                    hr[i][1] = x[i]
            x = hyperrect_sample(hr)
        samples.append(x)
    return samples
```

To achieve parallelism, I use Python's built-in library `multiprocessing`. In

each bracket of Hyperband, the list of configurations are trained and tested in parallel with the specified number of workers.

Chapter 6

Results and Analysis

6.1 Experimental Setup

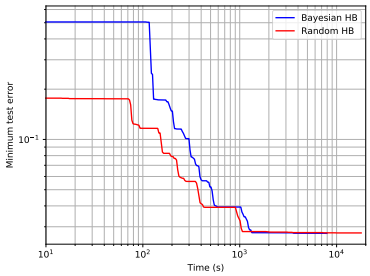
The experiments were run on the NTNU IDUN computing cluster (Själander et al. 2019). The cluster has more than 70 nodes and 90 GPGPUs. Each node contains two Intel Xeon cores, at least 128 GB of main memory, and is connected to an Infiniband network. Half of the nodes are equipped with two or more Nvidia Tesla P100 or V100 GPGPUs. Idun's storage is provided by two storage arrays and a Lustre parallel distributed file system.

Each run of each method had ten CPU cores (corresponding to ten workers in parallel) and 40 GB of RAM available, unless stated otherwise. The objective to minimize for all methods was the error rate of the machine learning model. The confidence intervals were obtained through bootstrapping (Efron and Tibshirani 1986).

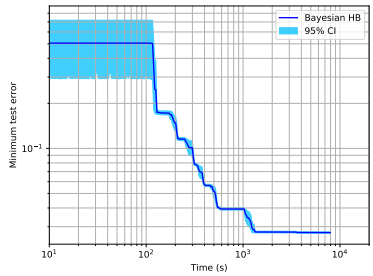
In this chapter, I will denote the combined method as described in chapter 5 as "Bayesian HB", and the standard Hyperband algorithm from chapter 4 as "Random HB". In all runs of Bayesian HB, 100000 samples were computed by the internal slice sampler; this choice was arbitrary, but chosen large in an attempt to diminish the effects of burn-in. This number may be large, but the time it takes to sample is many orders of magnitude smaller than the time it takes to train and test models in general.

6.2 Support Vector Machines

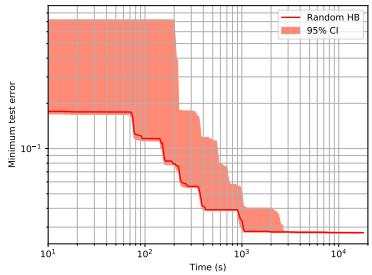
For this particular experiment, SVMs were trained on the MNIST training dataset (LeCun, Cortes, and Burges 2010), which are 28×28 greyscale images. The `SVC` class from Scikit-learn (Pedregosa et al. 2011) was used, which builds the classifier in a one-versus-one fashion to train and test the SVMs. The hyperparameters to optimize were `C` and `gamma`—`C` corresponding to λ in section 2.1.2, and `gamma` corresponding to γ in the RBF kernel described in section 2.1.3. The parameters were allowed to range between 10^{-10} and 10^{10}, identically as done by Falkner, Klein, and Hutter (2018). For the experiments using Random HB, the parameters were drawn from log-uniform distributions. For Bayesian HB, the parameters were modeled in log space by the GP as described in section 3.1.3. The resource R used was the number of available training data points ($R = 60000$), n_{max} was set to 243, and η was set to 3. This allowed for 6 brackets. Both the training dataset and the testing dataset were transformed by applying a `StandardScaler` object from Scikit-learn (Pedregosa et al. 2011) fitted on the training dataset; this ensures that each feature has zero mean and unit variance. For each method, ten experiments were performed. The results obtained are summarized in fig. 6.1a, fig. 6.1b and fig. 6.1c. Bayesian HB starts off worse in the beginning, but rapidly improves. Random HB may appear to converge slightly faster on average than Bayesian HB, but since the mean of Bayesian HB lies within the confidence interval of Random HB after Random HB has converged, we cannot say that Random HB truly converges faster.



(a) Mean minimum test error for Bayesian HB and Random HB as a function of time.



(b) Mean minimum test error and 95 percent confidence intervals for Bayesian HB.



(c) Mean minimum test error and 95 percent confidence intervals for Random HB.

Figure 6.1: Experimental results from training SVMs on the MNIST dataset.

6.3 Artificial Neural Networks

For these experiments, ANNs were trained on the MNIST dataset. The data was preprocessed by ensuring that each feature lied in the range $[0, 1]$. The hyperparameters to tune was the learning rate for stochastic gradient descent, the batch size, the number of hidden layers, the dropout for *all* hidden layers, and the size of *all* hidden layers. The hyperparameters and their bounds are summarized in fig. 6.1.

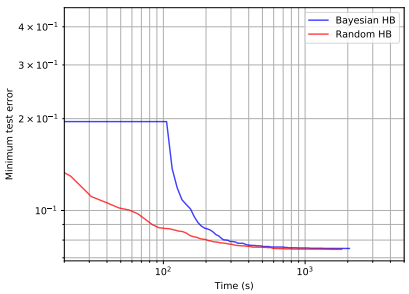
For the random configuration samples, the integer-valued parameters and

the dropout rate were drawn from uniform distributions. The learning rate was drawn from a log-uniform distribution. The learning rate was log-transformed inside Bayesian HB as described in section 3.1.3, and to obtain integer-valued parameters for the Bayesian method, these parameters were simply rounded to the closest integer.

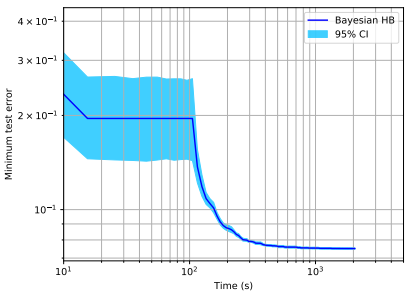
For these experiments, R was set to 81 epochs of stochastic gradient descent, and η was set to 3. The results from running both Bayesian HB and Random HB twenty times can be seen in fig. 6.2. Bayesian HB starts off much worse on average, but rapidly closes the gap. Both methods converge to the same value.

Hyperparameter	Type	Bounds
Learning rate	Float	$[10^{-6}, 10^{-2}]$
Dropout rate	Float	$[0, 0.5]$
Batch size	Integer	$[2^3, 2^8]$
# hidden layers	Integer	$[1, 5]$
Hidden layer size	Integer	$[2^4, 2^8$

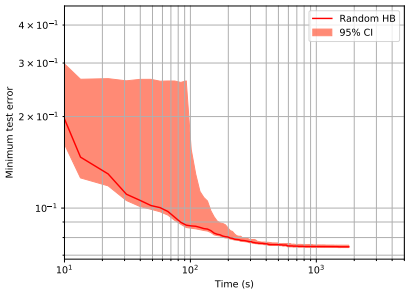
Table 6.1: Hyperparameters to optimize for the ANN experiments.



(a) Mean minimum test error for Bayesian HB and Random HB as a function of time.



(b) Mean minimum test error and 95 percent confidence intervals for Bayesian HB.



(c) Mean minimum test error and 95 percent confidence intervals for Random HB.

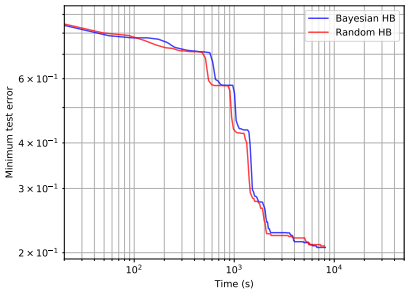
Figure 6.2: Experimental results obtained from training ANNs on the MNIST dataset. Twenty runs of both Bayesian HB and Random HB were performed.

6.4 Convolutional Neural Networks

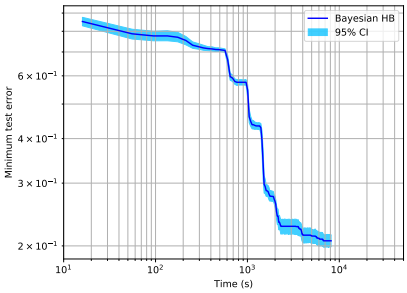
In this experiment, the parameters of a CNN were optimized; the CNNs were trained on the CIFAR10 dataset (Krizhevsky, G. Hinton, et al. 2009). The dataset was preprocessed to ensure that each feature was in the range $[0, 1]$. Details of the parameters and the architecture of the CNN can be found in appendix A. The CNNs were trained on a single Nvidia Tesla P100 GPU. The models were not trained in parallel, because I could not manage to make the experiments run in parallel within my time constraints. The experiment is still

meaningful, however, since the only difference now is that the models aren't trained in parallel.

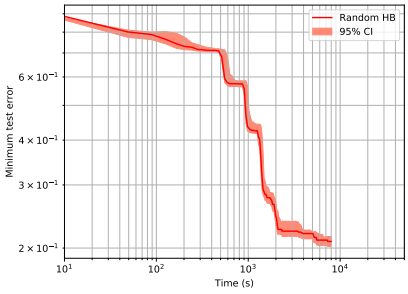
For these experiments, R was set to 81 epochs of stochastic gradient descent, and η was set to 3. The results of running both Bayesian HB and Random HB twenty times can be seen in fig. 6.3. As can be seen in fig. 6.3a, the convergence curves of both methods closely resemble one another, although neither method seems to have converged. A definite winner cannot be extrapolated from the data produced by the experiment, since the terminal mean of each method lies within the other's confidence interval.



(a) Mean minimum test error for Bayesian HB and Random HB as a function of time.



(b) Mean minimum test error and 95 percent confidence intervals for Bayesian HB.



(c) Mean minimum test error and 95 percent confidence intervals for Random HB.

Figure 6.3: Experimental results obtained from training CNNs on the CIFAR10 dataset. Twenty runs of both Bayesian HB and Random HB were performed.

Chapter 7

Conclusions and Future Work

The goal of this thesis was to determine whether Hyperband could be improved by incorporating GP-based Bayesian optimization. To reach that goal, the research questions in section 1.2 were constructed. The solution proposed to RQ1 is the method proposed in chapter 5. This is only one of a very large number of ways to combine the two methods, however. The probability of this proposed solution being the optimal is quite low, due to the sheer size of the space of possible solutions.

As for RQ2: Based on the results obtained in chapter 6, the combination of Bayesian optimization and Hyperband proposed in chapter 5 is not better or worse than vanilla Hyperband when rating methods by convergence rate. No method can be identified as having a better or worse convergence rate in any of the experiments.

There could be several reasons why the combined approach does not offer improvements over its vanilla counterpart. One possibility is that the search spaces considered here are too simple—if the search space isn't that complicated, i.e. locating good configurations is easy, then perhaps the combined method is never really leveraged since a good configuration could be located early by the random sampler. An idea for future work could then be to evaluate more complex search spaces.

Another possibility could be the fact that the combined method does not handle integral parameters in a manner that is sophisticated enough—integral

parameters are simply rounded to the closest integer by the Bayesian sampler of the combined method. A suggestion for handling integral parameters is the method proposed by Garrido-Merchán and Hernández-Lobato (2017).

Parameters that were assumed to introduce non-stationarity in the loss function were simply optimized in log space. This transformation is not necessarily correct for all kinds of parameters, so leveraging a more sophisticated method such as the one proposed by Snoek, Swersky, et al. (2014) would be an interesting idea for future work.

Finally, a reason that the combined approach does not perform so well could be caused by the fact that the method has parameters of its own, such as the number of slice (MCMC) samples and N_{min}. An analysis of the effect of these parameters was not performed in this thesis, but is a natural next step in evaluating the combined method's merit.

For the practitioner, I would suggest not bothering with the method proposed here. The implementation is much more complicated than just sampling at random, adds overhead, and does not provide any benefits in terms of configuration quality or speed. Random sampling also allows for flexible priors, leveraging *a priori* knowledge in a simple fashion.

Bibliography

Bergstra, James and Yoshua Bengio (2012). "Random search for hyper-parameter optimization". In: *Journal of machine learning research* 13.Feb, pp. 281–305.

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

Rasmussen, Carl Edward (2003). "Gaussian processes in machine learning". In: *Summer School on Machine Learning*. Springer, pp. 63–71.

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

Falkner, Stefan, Aaron Klein, and Frank Hutter (2018). "BOHB: Robust and efficient hyperparameter optimization at scale". In: *arXiv preprint arXiv:1807.01774*.

Bertrand, Hadrien et al. (2017). "Hyperparameter optimization of deep neural networks: Combining hyperband with Bayesian model selection". In: *Conférence sur l'Apprentissage Automatique*.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-vector networks". In: *Machine learning* 20.3, pp. 273–297.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

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

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814.

Bishop, Christopher M (2006). *Pattern recognition and machine learning*. springer.

Hecht-Nielsen, Robert (1992). "Theory of the backpropagation neural network". In: *Neural networks for perception*. Elsevier, pp. 65–93.

Bottou, Léon (2010). "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, pp. 177–186.

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

Rasmussen, Carl Edward and Christopher K. I. Williams (2006). *Gaussian Processes for Machine Learning*. MIT Press. ISBN: 978-0-262-18253-9. URL: http://www.gaussianprocess.org/gpml/chapters/RW.pdf.

Snoek, Jasper, Kevin Swersky, et al. (2014). "Input warping for Bayesian optimization of non-stationary functions". In: *International Conference on Machine Learning*, pp. 1674–1682.

Kushner, Harold J (1964). "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise". In: *Journal of Basic Engineering* 86.1, pp. 97–106.

Mockus, Jonas, Vytautas Tiesis, and Antanas Zilinskas (1978). "The application of Bayesian methods for seeking the extremum". In: *Towards global optimization* 2.117-129, p. 2.

Virtanen, Pauli et al. (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17, pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

Oliphant, Travis E (2006). *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Jamieson, Kevin and Ameet Talwalkar (2016). "Non-stochastic best arm identification and hyperparameter optimization". In: *Artificial Intelligence and Statistics*, pp. 240–248.

Wang, Jiazhuo, Jason Xu, and Xuejun Wang (2018). "Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning". In: *arXiv preprint arXiv:1801.01596*.

Bergstra, James S et al. (2011). "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems*, pp. 2546–2554.

Snoek, Jasper, Hugo Larochelle, and Ryan P Adams (2012). "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems*, pp. 2951–2959.

Neal, Radford M (2003). "Slice sampling". In: *Annals of statistics*, pp. 705–741.

Chib, Siddhartha and Edward Greenberg (1995). "Understanding the metropolis-hastings algorithm". In: *The american statistician* 49.4, pp. 327–335.

Själander, Magnus et al. (2019). *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. arXiv: 1912.05848 [cs.DC].

Efron, Bradley and Robert Tibshirani (1986). "Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy". In: *Statistical science*, pp. 54–75.

LeCun, Yann, Corinna Cortes, and CJ Burges (2010). "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2.

Krizhevsky, Alex, Geoffrey Hinton, et al. (2009). "Learning multiple layers of features from tiny images". In:

Garrido-Merchán, Eduardo C and Daniel Hernández-Lobato (2017). "Dealing with integer-valued variables in Bayesian optimization with Gaussian processes". In: *arXiv preprint arXiv:1706.03673*.

Chollet, François et al. (2015). *Keras*. `https://keras.io`.

Appendix A

Details of CNN Experiment

The architecture and parameters of the CNN experiment is best understood by inspecting the actual code. The implementation of the CNN class is built upon Keras (Chollet et al. 2015):

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD

class CNN:
    def __init__(self, learning_rate=None, f1=None, f2=None,
    f3=None, f4=None, drop1=None, drop2=None, drop3=None,
    dense_units=None, max_iter=None):
        model = Sequential()
        model.add(Conv2D(f1, (3, 3), padding='same',
                         input_shape=x_train.shape[1:]))
        model.add(Activation('relu'))
        model.add(Conv2D(f2, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(drop1))

        model.add(Conv2D(f3, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Conv2D(f4, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(drop2))
```

```
        model.add(Flatten())
        model.add(Dense(dense_units))
        model.add(Activation('relu'))
        model.add(Dropout(drop3))
        model.add(Dense(10))
        model.add(Activation('softmax'))

        model.compile(loss='categorical_crossentropy',
                      optimizer=SGD(lr=learning_rate),
                      metrics=['accuracy'])

        self.model = model

        self.epochs = max_iter
        print(self.epochs)


    def fit(self, x_train, y_train):
        self.model.fit(x_train, y_train, batch_size=256,
        epochs=self.epochs, verbose=0)

    def score(self, x_test, y_test):
        model_score = self.model.evaluate(x_test,
        y_test, verbose=0)[1]
        keras.backend.clear_session()
        return model_score
```

The code explicitly states the architecture of CNN. There are nine parameters that are optimized; `max_iter` is controlled by Hyperband. The parameters are fully specified by the `config_spec`:

```
    config_spec = [
        {
            "name": "f1",
            "type": "int",
            "scale": "uniform",
            "bounds": [2**3, 2**7]
        },
        {
            "name": "f2",
            "type": "int",
            "scale": "uniform",
            "bounds": [2**3, 2**7]
        },
        {
            "name": "f3",
            "type": "int",
            "scale": "uniform",
            "bounds": [2**3, 2**7]
```

```
        },
        {
            "name": "f4",
            "type": "int",
            "scale": "uniform",
            "bounds": [2**3, 2**7]
        },
        {
            "name": "drop1",
            "type": "float",
            "scale": "uniform",
            "bounds": [0, 0.5]
        },
        {
            "name": "drop2",
            "type": "float",
            "scale": "uniform",
            "bounds": [0, 0.5]
        },
        {
            "name": "drop3",
            "type": "float",
            "scale": "uniform",
            "bounds": [0, 0.5]
        },
        {
            "name": "learning_rate",
            "type": "float",
            "scale": "log",
            "bounds": [-7, -1]
        },
        {
            "name": "dense_units",
            "type": "int",
            "scale": "uniform",
            "bounds": [2**3, 2**9]
        }
    ]
```

In the experiment, the integral parameters were rounded to the closest integer by the Bayesian sampler.