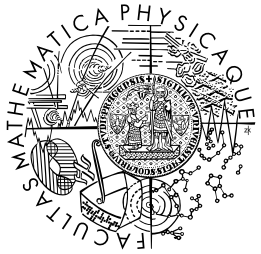

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Bc. Jakub Arnold

# Bayesian Optimization of Hyperparameters Using Gaussian Processes

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: RNDr. Milan Straka, Ph.D.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  signature of the author

Dedication.

Title: Bayesian Optimization of Hyperparameters Using Gaussian Processes

Author: Bc. Jakub Arnold

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Milan Straka, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Abstract.

Keywords: gaussian process bayesian optimization global optimization neural network

# Contents

# 1. Introduction

A machine learning algorithm is an algorithm that can learn from data. A commonly used definition is: "A computer program is said to learn form experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$" [Mitchell, 1997]. We only provide this definition for completeness and our following definitions will assume an intuitive translation to this definition.

Machine learning models are traditionally split into two categories, parametric and non-parametric. Parametric models have a fixed number of learnable parameters which are trained on a subset of the data called the training set. On the other hand, non-parametric models typically have a variable number of parameters which depends on the size of the data. This could mean that there is a parameter associated with each data point. Non-parametric does not mean that there are no parameters.

Most machine learning algorithms also have hyperparameters, which are parameters that the learning algorithm can not learn itself, and they usually control its behavior. A parameter could be chosen to be a hyperparameter because it would not be reasonable to infer its value from the training data. An example could be the type of optimizer being used to train a neural network (e.g. SGD or Adam [Kingma and Ba, 2014]). A machine learning practitioner would not consider the optimizer to be a property of the data distribution, and as a reasult treat it as a hyperparameter set externally, rather than trying to infer it.

MS: V americke anglictine je "e.g., ..." a "i.e., ...", v britske se tam ta carka nepise. Navzdory lekcim anglictiny, kde je British English ta "lepsi", se clanky pisou spis americkou; ale u thesis to nechavam na Tobe.

A parameter could also be set as a hyperparameter simply for practical reasons, where in theory the parameter could be learned from the data, but optimizing it directly would be too difficult. An example here could be the learning rate of a stochastic gradient descent optimizer. Even if automatic differentiation [Maclaurin et al., 2015] allows us to compute the gradient flow through arbitrary code, in practice it is not being used to compute the gradients of hyperparameters like the parameters of an optimizer, e.g. the learning rate. Instead, the learning rate is treated as a hyperparameter and is set ahead of time, or according to a fixed schedule. Even when early stopping or other heuristic methods are employed, it would still be treated as a hyperparameter from the perspective of the learning algorithm.

Bayesian statistics allows us to take a principled approach to setting hyperparameters. We would simply set a prior distribution on each hyperparameter and marginalize over them, but unfortunately this has two problems. The prior distribution almost always has parameters of its own, such as the rate parameter $\lambda$ of a Poisson distribution, or the concentration parameter vector $\boldsymbol{\alpha}$ of a Dirichlet distribution. We could go one step further and specify a prior on these parameters, which are often called hyperpriors. But the real problem of Bayesian methods is that the integral in the marginalization often ends up being intractable, and in the case of more complicated models as neural networks, we are already computationally bottlenecked and cannot use methods such as Markov-Chain Monte Carlo (MCMC) to approximate it.

There are however cases when the Bayesian approach does work. Either the model is small and simple enough so that the integral can be actually computed,

or its properties (such as conditional independence in LDA [Blei et al., 2003]) allow us to perform more efficient MCMC sampling, or the model could actually have an analytic solution to the marginalization. The last case is something we will make use of later on when we introduce Gaussian Processes.

Unfortunately, in the field of deep learning [Goodfellow et al., 2016] and neural networks, our models are almost always so large that just computing a point estimate of the parameters is close to our computational limits. For that reason we do not usually employ Bayesian methods, but look for alternative – less computationally heavy – approaches.

A simple approach to tune hyperparameter is either via random search, where each parameter is sampled randomly from a given prior distribution, or via grid search, where a fixed grid over the hyperparameter space is chosen, and then each point on the grid is evaluated, and the best parameter combination is chosen. The evaluation can be done using an arbitrary metric of performance, called the **objective function**. This could simply be the loss achieved by the model on the validation set, meaning we would create a function that takes the hyperparameters as its arguments, train the model on the training set, evaluate its performance on the validation set, and return that score as its value.

It is not difficult to see that neither of these approaches are optimal, as neither take into account the already computed values of the loss (objective). If the model is evaluated on one set of hyperparameters and achieves a high loss, we would like the hyperparameter tuning mechanism to take this into account, and possibly try a different combination. This process is similar to that of an agent trying to balance the exploration-exploitation tradeoff [Russell and Norvig, 2016]. On one hand, we would like to explore different combinations of hyperparameters and explore as much of the search space as possible. But once we find a combination with a high value of the objective function we would like to explore the space around that combination to exploit the high value and possibly find a close combination that is even better.

Therefore, we would like our hyperparameter optimization procedure to balance the exploration-exploitation trade-off, and take previous evaluation into account when choosing which point to evaluate next. Since the training of machine learning models – and neural networks in particular – can be computationally costly, we need to pick an optimization procedure that is sample efficient. Many modern deep learning models take days, weeks, or even months to train on high end hardware. To give a few examples, the recent OpenaI Five [ope] has consumed 800 PETAFLOPs-days over the course of 10 months. In comparison, a consumer grade GPU GTX 1080 Ti achieves just over 11 TFLOPs. A smaller and more realistic example would be the NVIDIA StyleGAN [Karras et al., 2018] trained on $1024 \times 1024$ images, which takes almost 7 days on $8\times$ Tesla V100 GPUs. In our experiments (see Chapter 6) we train a tagger and lemmatizer on Czech and English treebanks, where each evaluation takes about four GPU hours. Evaluating the objective function translates to training one such model from scratch with a given set of hyperparameters and obtaining an unbiased estimate of its performance on a validation set.

Even though we have the ability to evaluate the objective function, we have no way of computing its gradients, or obtain its analytic form. As a result we have to treat it as a black box and use optimization techniques which do not re-

quire either. But even with the smaller models we have just mentioned, it is easy to see that we cannot perform more than a few hundred of evaluations without spending unreasonable computational costs. This immediately disqualifies many of the common black box optimization techniques, such as evolution strategies or simulated annealing, which require on the order of thousands evaluations to converge [Golovin et al., 2017]. These methods do not model the objective function in any way, but rather search the target space directly. Evolutionary algorithms or simulated annealing never see the space as a whole, but rather perform some variant of stochastic hill climbing, where given enough stochasticity the methods can be considered to perform **global optimization**.

Bayesian optimization [Shahriari et al., 2016] is a black box optimization technique which utilizes a probabilistic model to take a set of evaluations of the objective function and computes a posterior over all possible functions given the observed data. As the next step, it computes an **acquisition function**, which is a function of the posterior, and represents the potential usefulness of the next sample. A popular example is the **expected improvement** function (Chapter 4), which is roughly defined as *the expected improvement over the maximum obtained so far*, or more formally

$$\mathrm{EI}(x) = \mathrm{E}\left[\max(0, f(x) - y_{\max})\right],$$

where $y_{\max}$ is the currently attained maximum. By sampling at the maximum of the acquisition function we obtain the point that is most likely going to help us the most. A key insight here is that the model is probabilistic. It does not simply fit a regression line through the data points and find the maximum. Instead, we fit a **Gaussian Process** (GP) which allows us to capture the uncertainty in the predictions. This uncertainty is taken into account by the acquisition function, and as a result it ends up balancing the exploration-exploitation trade-off by both taking into account the predicted value, and our uncertainty in that value.

A key benefit is that the GP is fitted to the whole space, as compared to the search based methods mentioned earlier. We can treat the GP as a surrogate model of the objective function, and optimize it instead, as evaluating the GP at any given point is usually orders of magnitude faster than evaluating the objective function.

## 1.1 Our Contributions

We implemented a tool for optimizing arbitrary programs' hyperparameters using Bayesian optimization, including a scheduler which runs the evaluations on a cluster, and a web interface visualizing the results. We do not provide any theoretical extensions to Bayesian optimization – the benefit is only in the implementation. This work however also serves as a thorough introduction to the theoretical background, specifically on GP. Understanding the theoretical aspects of GPs allows the user to interpret the behavior of the optimizer, as well as to better understand why some result visualizations might look the way they do.

Our implementation of Bayesian optimization utilizes the GPy library [GPy, since 2012] which implements the basic GP regression model. We chose to use the library mainly for the reasons of numerical stability. In theory, as we will show

later Chapter 3, implementing GP regression is simple for a small toy example. But with realistic data it is easy to run into numerical issues, some of which we will go over in the following Chapter 2. Apart from numerically stable code, the GPy allows for more control over the hyperparameters of the GP using constrained optimization.

Fitting a GP model and optimizing the acquisition function is just the beginning. A non-trivial amount of the work is devoted to evaluating the function in a flexible way. In our case, we expect the user to provide a script which encapsulates the function, accepts its parameters in the form of command line arguments, and prints the result of the function on its standard output. This approach allows us to run the evaluations in parallel, or even run them on a cluster. The implementation is flexible enough so that a user could provide their own way of running the evaluations, should they have specific needs. The experimental data is also stored in an easy to access text format, with command line utilities that allow the user a fine grained control over the experiment.

Lastly, running real-life experiments can be a time consuming process, and having the ability to monitor the process and interfere if needed is an important feature. This is why we provide a web interface that can visualize the progress of the optimization. The user can explore the evaluated samples, as well as the regression model at any point in time during the optimization.

# 2. Bayesian Optimization

In this chapter we explore Bayesian optimization from a higher level perspective. We will describe the steps of the algorithm and give more motivation for choosing GPs as a probabilistic model. We also explain the difference between hyperparameter tuning and architecture search, which, even though similar at first, are completely different tasks. Lastly, we present some examples of related work, both from the point of hyperparameter tuning, as well as related areas, such as aut.

Consider the problem of optimizing an arbitrary continuous function $f : \mathcal{X} \to \mathbb{R}$ where $\mathcal{X} \subset \mathbb{R}^d, d \in \mathbb{N}$. We call $f$ the *objective function* and treat it as a black box, making no other assumptions on its analytical form, or on our ability to compute its derivatives. Our goal is to find the global minimum $\boldsymbol{x}_{\text{opt}}$ over the set $\mathcal{X}$, that is

$$\boldsymbol{x}_{\text{opt}} = \arg\min_{\boldsymbol{x} \in \mathcal{X}} f(\boldsymbol{x}).$$

We also assume that the evaluation of $f$ is expensive, as the goal of Bayesian optimization is to find the optimum in as few evaluations as possible. Consider the case when evaluating $f$ means performing a computation that is not only time consuming, but for example also costs a lot of money. We might only have a fixed budget which puts a hard limit on the number of evaluations we can perform. If the function can be evaluated cheaply, other global optimization approaches such as simulated annealing or evolution strategies could potentially yield better results [Golovin et al., 2017].

Bayesian optimization techniques are some of the most efficient approaches in terms of the number of function evaluations required. Much of the efficiency stems from the ability to incorporate prior belief about the problem and to trade of exploration and exploitation of the search space [Brochu et al., 2010].

It is called bayesian because it combines the prior knowledge $p(f)$ about the function together with the data in the form of the likelihood $p(x|f)$ to formulate a posterior distribution on the set of possible functions $p(f|x)$. We will use the posterior distribution to figure out which point should be evaluated next to give a likely improvement over the currently obtained maximum. This improvement is defined by an **acquisition function**, which represents our optimization objective. A simple example of an acquisition function is the **probability of improvement**, which simply represents the probability of improving our objective compared to the previously achieved maximum. We will show a few other examples of acquisition functions in a later Section 4.1.

The optimization procedure is sequential, using a bayesian posterior update at each step, refining its model as more data are available. At each step the we maximize the acquisition function in order to obtain the next sample point $x_{i+1}$. We then evaluate $f(x_{i+1})$ to obtain $y_{i+1}$, and incorporate it into the dataset. This processs is repeated for as many evaluations as we can perform. See algorithm 1 for pseudocode.

At its core, Bayesian optimization only requires two things. A probabilistic regression model which combines prior beliefs $p(f)$ with the data. And an acquisition function which describes the optimality of the next sampling point.

Initialize $\boldsymbol{x}_1$ randomly and evaluate $y_1 = f(\boldsymbol{x}_1), \boldsymbol{\mathcal{D}}_1 = (\boldsymbol{x}_1, y_1)$.
**for** $i = 1, 2, \ldots$ **do**
    Find $\boldsymbol{x}_{i+1}$ by maximizing the acquisition function.
    Evaluate $y_{i+1} = f(\boldsymbol{x}(x_{i+1}))$.
    Add the sample to the dataset $\boldsymbol{\mathcal{D}}_{i+1} = \boldsymbol{\mathcal{D}}_i \cup (\boldsymbol{x}_{i+1}, y_{i+1})$.
    Update the probabilistic model.
**end**

**Algorithm 1:** Bayesian Optimization, Brochu et al. [2010]

## 2.1 Prior Distribution over Functions

Bayesian methods by definition require a prior distribution over the quantity of interest. Since we are building a probabilistic model over functions, we need to obtain a prior distribution over functions, which will capture our general beliefs about the properties of the objective function. For example, if we knew that our function was periodic, we would want a prior distribution over periodic functions. But in the case of hyperparameter optimization we will generally only consider continuous functions.

We will follow the general consensus of using a GP as a prior distribution over functions [Brochu et al., 2010], as it provides many nice theoretical properties, as well as tractable posterior inference. Other models, such as random forests are also possible (see Section 2.2).

A GP is an extension of the multivariate Gaussian distribution to infinitely dimensional random variables. Just as a multivariate Gaussian can be thought of as a distribution over vectors, a GP can be thought of as a distribution over infinitely dimensional vectors, which when indexed by the real numbers are equivalent to functions. A GP assumes that any finite subset $\mathrm{x} = (x_1, \ldots, x_n)$ is jointly Gaussian with some mean $m(\mathrm{x})$ and covariance $\Sigma(\mathrm{x})$. A GP is defined by its mean function $m$ and covariance function $\kappa$ [Murphy and Bach, 2012]. We write

$$\mathcal{GP}(m(\mathrm{x}), \kappa(\mathrm{x}, \mathrm{x}')).$$

By convention, we assume that the prior mean is a constant zero function, that is $m(x) = 0$. Since the data is often normalized in practice, this does not reduce the flexibility of the model. The power of a GP comes from its covariance function, which for any two points $\mathrm{x}_i$ and $\mathrm{x}_j$ defines their covariance $\kappa(\mathrm{x}_i, \mathrm{x}_j)$. If $\mathrm{x}_i$ and $\mathrm{x}_j$ have a high covariance, the values of the function at those points will be similar.

Figure 2.1 shows an example of GP regression in one dimension. Because GP is a non-parametric model it uses the whole dataset $\mathcal{D}$ in order to compute the posterior parameters. A theoretical benefit of this approach, as compared to using a parametric model like a random forest, is that we reduce the number of ways our model can underfit or overfit the data. Because we assume the mean to be zero, our only parameter is the kernel function $\kappa$. As we will explain later in Subsection 3.4.3 we restrict ourselves even further to the Matérn covariance function, which only has two hyperparameters.

Because the model is non-parametric, it can still fit any arbitrary dataset we give it, but its flexibility in terms of overfitting is controlled only via the kernel

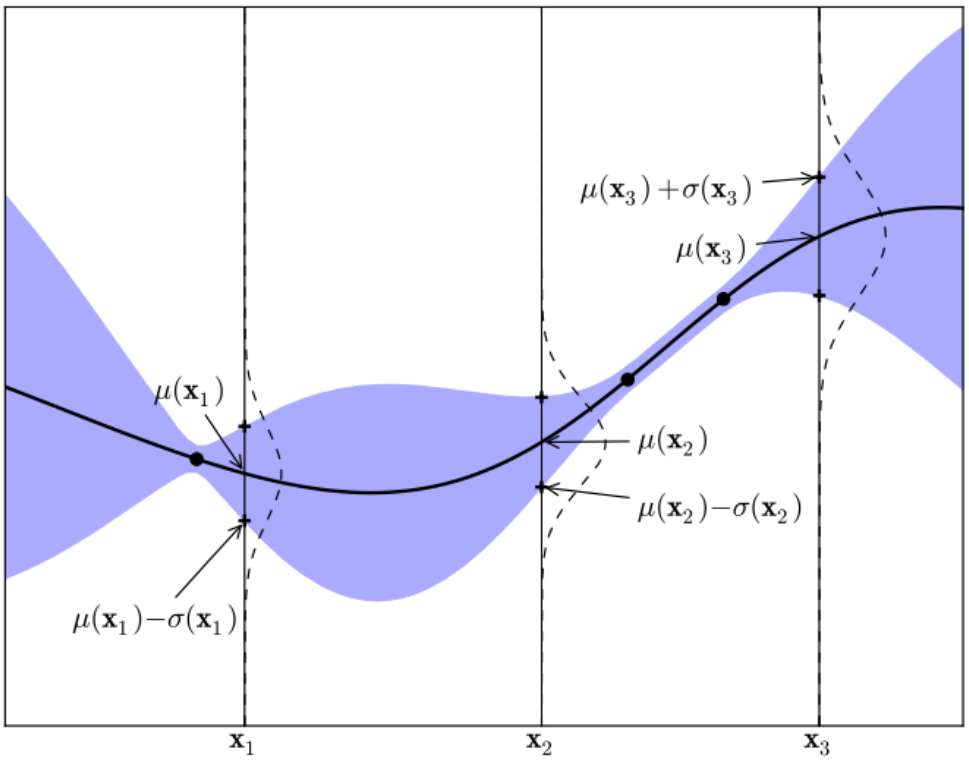


Figure 2.1: A GP regression fit to three data points in 1D, shown as small black circles. The black line signifies the mean prediction, while the purple filled area shows one standard deviation at each point. The three points marked as $x_1, x_2, x_3$ are where we are computing the posterior, that is $p(x_1, x_2, x_3 \mid \mathcal{D})$. These points together have a multivariate Gaussian distribution, with a marginal 1D distribution shown vertically at each point. In order to draw a plot like this one we compute the posterior over a fine grid on the $X$ axis, and plot the mean parameter and the diagonal of the covariance matrix, giving us the standard deviation. Image source: Brochu et al. [2010]

parameters. As an added benefit we can visualize the change in kernel parameters over time during Bayesian optimization in order to get further insight into how the model is fitting the data, as shown in Subsection 5.5.1.

Intuitively, it is often useful to think of a GP as a function, which instead of returning a scalar $f(x)$ returns the mean and standard deviation of a Gaussian distribution over the possible values, centered at $x$. We leave a formal treatment of GPs until Chapter 3.

## 2.2 Hyperparameter vs. Architecture Search

Using our earlier definition, any parameter which the model does not learn on its own could be considered a hyperparameter. This definition is broad enough to allow a lot of flexibility, but some hyperparameters are better for the framework of Bayesian optimization than others. Discrete and categorical hyperparameters require special consideration. Bayesian optimization itself is flexible in the sense that it allows for an arbitrary probabilistic model, but the specific choice does matter when we consider discrete values. In the case of GPs, the model itself does not have the ability to directly work with anything but continuous real valued vectors.

There has been some recent work [Garrido-Merchán and Hernández-Lobato, 2017] showing better approaches for handling integer valued variables. This is done by simply rounding the appropriate values before computing their covariance. As the kernel will see the values as equivalent, their covariance will be maximized, and the GP will be forced to predict a constant value over each integer region. While this approach does help a little bit with integer valued variables, it does not work for categorical (nominal) variables, which lack any form of ordering. If we simply treat them as integers, the GP prior will enforce relationships which do not exist.

An alternative approach to solving the problem with categorical variables is to use a different model than GPs, namely random forests [Shahriari et al., 2016]. Their main advantage is the ability to naturally handle various types of data. We however do not explore random forests in this work, as many of the hyperparameters of interest when tuning neural networks are either continuous or integer valued. Categorical variables, such as activation functions, are better suited to be tweaked as part of neural architecture search [Zoph et al., 2017].

Regardless of the probabilistic model, categorical variables cause many immediate problems. Consider the choice between SGD with momentum [Ruder, 2016] and Adam [Kingma and Ba, 2014]. While SGD with momentum has a *momentum* hyperparameter, Adam does not, but it has its own two extra hyperparameters, $\beta_1$ and $\beta_2$. This gives us two different sets of mutually exclusive hyperparameters. Bayesian optimization however does not have any natural way of handling problems like this. Even if we did externally enforce two different modes based on which categorical values was chosen, it would essentially be the same as running two experiments in parallel, one with SGD, and one with Adam. Another issue arises in visualization, which is one of the goals of this work. Inspecting the samples from two or more different modes of the network at once is challenging at best, and with multiple categorical variables the problem grows exponentially.

For these reasons, we chose to not provide direct support for categorical variables, apart form converting them to integer variables with an ordering and treating them as such. Some categorical variables can be optimized by simply treating them as fixed within a particular experiment, and then running multiple instances of that experiment with a different value each. Other categorical variables, such as the types of layers, activation functions, or even the connections between modules, are better left for the framework of neural architecture search, which treats them in a principled way.

## 2.3 Related work

There are a few notable mentions of related work in the area of tuning hyperparameters of neural networks. The main motivation for this work, is Google Vizier [Golovin et al., 2017], which is a proprietary service at Google used for black box optimization. There also exist a few implementations of Bayesian optimization. Spearmint [Group, 2014] is the most fully featured one, but comes with a very restrictive license, and does not perform any visualization of the results. GPyOpt [authors, 2016], RoBO [Klein et al., 2017] and scikit-optimize [Head et al., 2018] are the most notable libraries for Bayesian optimization, but they only provide the basic optimization loop for Bayesian optimization, and do not handle long running experiments in a possibly distributed environment.

An important mention is the AutoML [aut], which is an attempt at automating many aspects of Machine Learning. Many different approaches are being tried, for example the TPOT [Olson et al., 2016] library uses evolutionary algorithms to not only tune hyperparameters, but also perform architecture search on Scikit-Learn algorithms, including building feature pre-processing pipelines, dimensionality reduction and feature elimination.

Unfortunately, a commonly shared problem of these higher level approaches is the explosion of the search space. The more of the ML problem is handed over to the search procedure, the bigger the search space gets, and at some point one has to make certain trade-offs. The benefit of libraries as TPOT is their ease of use on small problems where the model can be trained within a few minutes. But as is the goal fo this thesis, we are interested in tuning hyperparameters of models which can take hours or even days to train, and in such case we want to be as efficient as possible.

Commercial services are also becoming more and more popular, both for AutoML and just for hyperparameter optimization. Apart from the Google Vizier service [Golovin et al., 2017] there also exist others, for example SigOpt [sig, b] provides a cloud based solution to hyperparameter tuning using Bayesian optimization. Amazon SageMaker [ama] is yet another commercially available cloud based hyperparameter tuning service based on Bayesian optimization.

# 3. Gaussian Processes

This chapter goes into the technical details of the Gaussian distribution and its extension the Gaussian Process (GP). We think it is important to have at least a basic understanding of the underlying math to make intuitive claims about the behavior of the model, especially since GPs are a bit different from other parametric machine learning models.

Since our objective is bayesian optimization, we only derive the properties necessary for its implementation. Specifically, we are interested in the conditional and marginal distributions of a multivariate Gaussian. The conditional Gaussian distribution allows us to compute the posterior $p(f|x)$ at an arbitrary point, and the marginal allows us to fit a GP regression model to each hyperparameter separately for additional visualization.

Let us now continue with a more rigorous treatment of the Gaussian distribution. For a more thorough treatment see Bishop [2016] and Murphy and Bach [2012].

**Definition 1.** *A random variable* X *has a* **univariate Gaussian distribution**, *written as* $\mathrm{X} \sim \mathcal{N}(\mu, \sigma^2)$, *when its density is*

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}.$$

*The parameters* $\mu$ *and* $\sigma$ *are its* mean *and* standard deviation.

**Definition 2.** *We say* X *has a* **degenerate Gaussian distribution** *when* $\mathrm{X} \sim \mathcal{N}(\mu, 0)$, *which formally becomes a Dirac delta function centered around* $\mu$.

**Definition 3.** *A random variable* $\mathbf{X} \in \mathbb{R}^n$ *has a* **multivariate Gaussian distribution** *if any linear combination of its components is a univariate Gaussian, i.e.* $\boldsymbol{a}^T\mathrm{X} = \Sigma_{i=1}^n \boldsymbol{a}_i \mathbf{X}_i$ *is a Gaussian for all* $\boldsymbol{a} \in \mathbb{R}^n$. *We then write* $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ *where* $\mathbb{E}[\mathbf{X}_i] = \mu_i$ *and* $\mathit{cov}(\mathbf{X}_i, \mathbf{X}_j) = \boldsymbol{\Sigma}_{ij}$.

*Remark.* The parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ uniquely determine the distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

**Definition 4.** *A random variable* $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ *has a* **degenerate multivariate Gaussian distribution** *if* $\det \boldsymbol{\Sigma} = \mathbf{0}$.

*Remark.* Given a random variable $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, random variables $\mathbf{X}_1, \dots, \mathbf{X}_n$ are independent with distributions $\mathbf{X}_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$ if and only if $\mu = (\mu_1, \dots, \mu_n)$ and $\Sigma = \mathit{diag}(\sigma_1^2, \dots, \sigma_n^2)$.

**Theorem 1.** *If a random variable* $\mathbf{X} \in \mathbb{R}^n$ *is a multivariate Gaussian, then* $\mathrm{X}_i, \mathrm{X}_j$ *are independent if and only if* $\mathit{cov}(\mathrm{X}_i, \mathrm{X}_j) = 0$. *Note that his is not true for any random variable, as it is a special property of the multivariate Gaussian.*

*Proof.* TODO □

**Theorem 2.** *A Gaussian random variable* $\mathrm{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ *has a density iff it is non-degenerate (i.e.* $\det \boldsymbol{\Sigma} \neq 0$, *alternatively* $\boldsymbol{\Sigma}$ *is positive-definite). And in this case, the density is*

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma})}} \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\} \tag{3.1}$$

*Remark.* The normalizing constant in the denominator is also often in an alternate form as

$$\det(2\pi\mathbf{\Sigma}) = (2\pi)^n \det(\mathbf{\Sigma})$$

which follows from basic determinant properties. Alternatively we can also put the square root in the exponent $(2\pi)^{n/2}(\det \mathbf{\Sigma})^{1/2}$.

*Remark.* A special case of the multivariate gaussian is when $n = 1$, then Note that if $n = 1$, then $\mathbf{\Sigma} = \sigma^2$, meaning $cov(X,X) = \sigma^2$ , $\mathbf{\Sigma}^{-1} = \frac{1}{\sigma^2}$, and hence the multivariate Gaussian formula becomes the univariate one

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}. \tag{3.2}$$

## 3.1 Sampling

Even not of immediate interest for Bayesian optimization, we will shortly show how to generate samples from a multivariate Gaussian, as this can be useful for visualization purposes with GPs.

**Theorem 3.** *Given a random variable* $\mathbf{X}$ *with* $cov[\mathbf{X}] = \mathbf{\Sigma}$*, it follows from the definition of covariance that* $cov[\mathbf{AX}] = \mathbf{A\Sigma A}^T$*.*

*Proof.*

$$\begin{aligned}
cov[\mathbf{AX}] &= E[(\mathbf{AX} - E[\mathbf{AX}])(\mathbf{AX} - E[\mathbf{AX}])^T] && (3.3)\\
&= E[(\mathbf{AX} - \mathbf{A}E[\mathbf{X}])(\mathbf{AX} - \mathbf{A}E[\mathbf{X}])^T] && (3.4)\\
&= E[\mathbf{A}(\mathbf{X} - E[\mathbf{X}])(\mathbf{X} - E[\mathbf{X}])^T\mathbf{A}^T] && (3.5)\\
&= \mathbf{A}E[(\mathbf{X} - E[\mathbf{X}])(\mathbf{X} - E[\mathbf{X}])^T]\mathbf{A}^T && (3.6)\\
&= \mathbf{A}cov[\mathbf{X}]\mathbf{A}^T && (3.7)\\
&= \mathbf{A}\Sigma\mathbf{A}^T && (3.8)
\end{aligned}$$

□

**Theorem 4.** *Given a random variable* $\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ *and a positive-definite matrix* $\mathbf{\Sigma}$ *with a Cholesky decomposition* $\mathbf{\Sigma} = \mathbf{LL}^T$*, then*

$$\mathbf{LX} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}). \tag{3.9}$$

*Proof.* We can immediately use equation 3.8.

$$\mathbf{LX} \sim N(0, \mathbf{LIL}^T) = N(0, \mathbf{LL}^T) = N(0, \mathbf{\Sigma}) \tag{3.10}$$

□

**Theorem 5.** *Any affine transformation of a Gaussian is a Gaussian. In particular*

$$\mathrm{X} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma}) \implies \mathbf{A}\mathrm{X} + \boldsymbol{b} \sim \mathcal{N}(\mathbf{A}\boldsymbol{\mu} + \boldsymbol{b}, \mathbf{A\Sigma A}^T)$$

*for any* $\boldsymbol{\mu} \in \mathbf{R}^n, \mathbf{\Sigma} \in \mathbf{R}^{n\times n}$ *positive semi-definite, and any* $\mathbf{A} \in \mathbf{R}^{m\times n}, \boldsymbol{b} \in \mathbf{R}^m$*. We call this the* **affine property** *of a Gaussian.*

*Proof.* Follows from the linearity of expectation together with Equation 3.9. □

Since samples from $\mathcal{N}(\mathbf{0}, \mathbf{I})$ can be generated independently, using the affine property we can generate samples from an arbitrary multivariate Gaussian. All that is required is a procedure for Cholesky decomposition, and a way of generating independent samples from a univariate Gaussian, which can be achieved using the Box-Muller transform [Box, 1958].

## 3.2 Geometric Properties

If $\mathbf{\Sigma}$ is positive-definite, then $\mathbf{Y} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma})$ implies $\mathbf{A}^{-1}(\mathbf{Y} - \boldsymbol{\mu}) \sim \boldsymbol{\mathcal{N}}(0, \mathbf{I})$ where $\mathbf{\Sigma} = \mathbf{A}\mathbf{A}^T$. The random variable $\mathbf{A}^{-1}(\mathbf{Y} - \boldsymbol{\mu})$ has a spherical shape in $n$-dimensional space.

Looking further at the density formula for a multivariate Gaussian (Equation 3.1) the term $(\boldsymbol{x}-\boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})$ is called the Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{\mu}$. If we consider $\boldsymbol{\mu}$ a constant, we can also view it as a quadratic form in $x$. When $\mathbf{\Sigma}$ is an identity matrix, the Mahalanobis distance reduces to Euclidean distance. In general, it can be thought of as a distance on a hyper-ellipsoid. Let us now derive some intuition for this.

Since $\mathbf{\Sigma}$ is a covariance matrix, we know it is positive definite, and we can perform its eigendecomposition to get $\mathbf{\Sigma} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$, where $\mathbf{U}$ is an orthogonal matrix of eigenvectors, and $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues. Basic matrix algebra gives us

$$\mathbf{\Sigma}^{-1} = (\mathbf{U}^T)^{-1}\mathbf{\Lambda}^{-1}\mathbf{U}^{-1} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^T = \sum_{i=1}^{D} \frac{1}{\lambda_i}\boldsymbol{u}_i\boldsymbol{u}_i^T,$$

where the second to last equality comes from $\mathbf{U}$ being orthogonal ($\mathbf{U}^{-1} = \mathbf{U}^T$). Substituting this in the Mahalanobis distance we get

$$(\boldsymbol{x} - \boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) = (\boldsymbol{x} - \boldsymbol{\mu})^T \left( \sum_{i=1}^{D} \frac{1}{\lambda_i}\boldsymbol{u}_i\boldsymbol{u}_i^T \right) (\boldsymbol{x} - \boldsymbol{\mu}) \tag{3.11}$$

$$= \sum_{i=1}^{D} (\boldsymbol{x} - \boldsymbol{\mu})^T \frac{1}{\lambda_i}\boldsymbol{u}_i\boldsymbol{u}_i^T(\boldsymbol{x} - \boldsymbol{\mu}) \tag{3.12}$$

$$= \sum_{i=1}^{D} \frac{y_i^2}{\lambda_i} \tag{3.13}$$

where $y_i = u_i^T(\boldsymbol{x} - \boldsymbol{\mu})$ which has exactly the same form as a $D$ dimensional ellipse. From this we conclude that the contour lines of a multivariate Gaussian will be elliptical, where the eigenvectors determine the orientation of the ellipse, and the eigenvalues determine the length of the principal axes [Bishop, 2016].

## 3.3 Conditional and Marginal Gaussian Distribution

In this section we derive the conditional $p(x_1 \,|x_2)$ and marginal $p(x_1)$ for a given joint distribution $p(x_1, x_2)$. One of the interesting properties of a multivariate

Gaussian is that both the conditional and the marginal are also Gaussian, and we can easily compute their parameters in closed from based on the parameters of the joint distribution.

Before we derive the conditional and marginal distributions, let us state the partitioned inverse formula without proof.

**Theorem 6** ([Murphy and Bach, 2012])**.** *Consider a partitioned matrix*

$$\mathbf{M} = \begin{bmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{bmatrix} \tag{3.14}$$

*where we assume* $\mathbf{E}$ *and* $\mathbf{H}$ *are invertible. We have*

$$\mathbf{M}^{-1} = \begin{bmatrix} (\mathbf{M}/\mathbf{H})^{-1} & -(\mathbf{M}/\mathbf{H})^{-1}\mathbf{F}\mathbf{H}^{-1} \\ -\mathbf{H}^{-1}\mathbf{G}(\mathbf{M}\mathbf{H})^{-1} & \mathbf{H}^{-1} + \mathbf{H}^{-1}\mathbf{G}(\mathbf{M}/\mathbf{H})^{-1}\mathbf{F}\mathbf{H}^{-1} \end{bmatrix} \tag{3.15}$$

$$= \begin{bmatrix} \mathbf{E}^{-1} + \mathbf{E}^{-1}\mathbf{F}(\mathbf{M}/\mathbf{E})^{-1}\mathbf{G}\mathbf{E}^{-1} & -\mathbf{E}^{-1}\mathbf{F}(\mathbf{M}/\mathbf{E})^{-1} \\ -(\mathbf{M}/\mathbf{E})^{-1}\mathbf{G}\mathbf{E}^{-1} & (\mathbf{M}/\mathbf{E})^{-1} \end{bmatrix} \tag{3.16}$$

*where*

$$\mathbf{M}/\mathbf{H} = \mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G} \tag{3.17}$$

$$\mathbf{M}/\mathbf{E} = \mathbf{H} - \mathbf{G}\mathbf{E}^{-1}\mathbf{F} \tag{3.18}$$

*is called the* **Schur complement**.

*Proof.* Since the proof is rather technical and only consists of applying the LDU decomposition and many algebraic manipulations, we leave it out and refer the reader to Murphy and Bach [2012] for details. □

### 3.3.1 Conditional Distribution is a Gaussian

Suppose $\boldsymbol{x}$ is a $D$-dimensional random vector with a multivariate Gaussian distribution $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and that $\boldsymbol{x}$ is partitioned into two vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ such that

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} \tag{3.19}$$

We also partition the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ into a block matrix, and name the inverse of the covariance matrix $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$, which will simplify a few of the equations that follow. We will derive the exact form of $\boldsymbol{\Lambda}$ and of its individual blocks later in this section. For now we simply use the fact that $\boldsymbol{\Sigma}$ is positive-definite, and thus it is invertible. The matrix $\boldsymbol{\Lambda}$ is also known as a **precision matrix**.

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}, \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{bmatrix} \tag{3.20}$$

Note that since $\boldsymbol{\Sigma}$ is a symmetric matrix, ${\boldsymbol{\Sigma}_{12}}^T = \boldsymbol{\Sigma}_{21}$, and similarly ${\boldsymbol{\Lambda}_{12}}^T = \boldsymbol{\Lambda}_{21}$. Similarly, $\boldsymbol{\Sigma}_{11}$, $\boldsymbol{\Sigma}_{22}$, $\boldsymbol{\Lambda}_{11}$, and $\boldsymbol{\Lambda}_{22}$ are all symmetrical.

Before we derive the parameters of the conditional, we show that the conditional distribution $p(x_1|x_2)$ is a Gaussian. To do this, we take the joint distribution $p(x_1, x_2)$ and fix the value of $x_2$ [Bishop, 2016]. Using the definition of conditional probability $p(x_1, x_2) = p(x_1|x_2)p(x_2)$ we can see that after fixing the value of $x_2$, $p(x_2)$ is simply a normalization constant, and the remaining term $p(x_1|x_2)$ is a function of $x_1$ which together with the normalization constant gives us the conditional probability distribution on $x_1$. We now use the partitioned form of the multivariate Gaussian defined by equation 3.20 to show that $p(x_1|x_2)$ is actually a Gaussian.

Let us begin by looking at the exponent in equation 3.1:

$$-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Lambda}(\boldsymbol{x}-\boldsymbol{\mu}) = -\frac{1}{2}\left(\begin{bmatrix}\boldsymbol{x}_1\\\boldsymbol{x}_2\end{bmatrix}-\begin{bmatrix}\boldsymbol{\mu}_1\\\boldsymbol{\mu}_2\end{bmatrix}\right)^T\begin{bmatrix}\boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12}\\\boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22}\end{bmatrix}\left(\begin{bmatrix}\boldsymbol{x}_1\\\boldsymbol{x}_2\end{bmatrix}-\begin{bmatrix}\boldsymbol{\mu}_1\\\boldsymbol{\mu}_2\end{bmatrix}\right) \tag{3.21}$$

$$= -\frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1-\boldsymbol{\mu}_1\\\boldsymbol{x}_2-\boldsymbol{\mu}_2\end{bmatrix}^T\begin{bmatrix}\boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12}\\\boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22}\end{bmatrix}\begin{bmatrix}\boldsymbol{x}_1-\boldsymbol{\mu}_1\\\boldsymbol{x}_2-\boldsymbol{\mu}_2\end{bmatrix} \tag{3.22}$$

To make the next few equations easier to follow we set $\boldsymbol{y}_1 = \boldsymbol{x}_1 - \boldsymbol{\mu}_1$ and $\boldsymbol{y}_2 = \boldsymbol{x}_2 - \boldsymbol{\mu}_2$.

$$-\frac{1}{2}\begin{bmatrix}\boldsymbol{y}_1\\\boldsymbol{y}_2\end{bmatrix}^T\begin{bmatrix}\boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12}\\\boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22}\end{bmatrix}\begin{bmatrix}\boldsymbol{y}_1\\\boldsymbol{y}_2\end{bmatrix} = \tag{3.23}$$

$$= -\frac{1}{2}\begin{bmatrix}\boldsymbol{y}_1\boldsymbol{\Lambda}_{11}+\boldsymbol{y}_2\boldsymbol{\Lambda}_{21}\\\boldsymbol{y}_1\boldsymbol{\Lambda}_{12}+\boldsymbol{y}_2\boldsymbol{\Lambda}_{22}\end{bmatrix}^T\begin{bmatrix}\boldsymbol{y}_1\\\boldsymbol{y}_2\end{bmatrix} \tag{3.24}$$

$$= -\frac{1}{2}\left(\boldsymbol{y}_1^T\boldsymbol{\Lambda}_{11}\boldsymbol{y}_1+\boldsymbol{y}_2^T\boldsymbol{\Lambda}_{21}\boldsymbol{y}_1+\boldsymbol{y}_1^T\boldsymbol{\Lambda}_{12}\boldsymbol{y}_2+\boldsymbol{y}_2^T\boldsymbol{\Lambda}_{22}\boldsymbol{y}_2\right) \tag{3.25}$$

$$\begin{aligned} = &-\frac{1}{2}(\boldsymbol{x}_1-\boldsymbol{\mu}_1)^T\boldsymbol{\Lambda}_{11}(\boldsymbol{x}_1-\boldsymbol{\mu}_1)+\\ &-\frac{1}{2}(\boldsymbol{x}_2-\boldsymbol{\mu}_2)^T\boldsymbol{\Lambda}_{21}(\boldsymbol{x}_1-\boldsymbol{\mu}_1)+\\ &-\frac{1}{2}(\boldsymbol{x}_1-\boldsymbol{\mu}_1)^T\boldsymbol{\Lambda}_{12}(\boldsymbol{x}_2-\boldsymbol{\mu}_2)+\\ &-\frac{1}{2}(\boldsymbol{x}_2-\boldsymbol{\mu}_2)^T\boldsymbol{\Lambda}_{22}(\boldsymbol{x}_2-\boldsymbol{\mu}_2) \end{aligned} \tag{3.26}$$

We see that this is a quadratic form in $x_1$, and hence the corresponding conditional distribution $p(x_1|x_2)$ will be Gaussian. Because we know $p(x_1, x_2) = p(x_1|x_2)p(x_2)$ and that both $p(x_1, x_2)$ and $p(x_1|x_2)$ are multivariate Gaussians, fixing the value of $x_1$ means that $p(x_1|x_2)$ as a function of $x_2$ is just a normalization constant, and $p(x_2)$ must have the same form as $p(x_1, x_2)$ and therefore is also a multivariate Gaussian.

tady ten marginal nevim jiste

TODO: x2 je spatne (marignal)

Because the Gaussian distribution is completely defined by its mean and covariance, we do not need to figure out the value of the normalization constant. We simply have to derive the equations for $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

We continue with the proof from Murphy and Bach [2012]. We will make use of the partitioned matrix inverse theorem Equation 3.16.

At this point we know that the joint distribution factors into two multivariate Gaussians, that is

$$p(x_1, x_2) = p(x_1 | x_2) p(x_2) \tag{3.27}$$
$$= \mathcal{N}(x_1 | \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \mathcal{N}(x_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \tag{3.28}$$

and we only need to infer their parameters. To make the equations more readable, we again define

$$\boldsymbol{y}_1 = \boldsymbol{x}_1 - \boldsymbol{\mu}_1 \tag{3.29}$$
$$\boldsymbol{y}_2 = \boldsymbol{x}_2 - \boldsymbol{\mu}_2. \tag{3.30}$$

We then simply take the block definition of a multivariate Gaussian and multiply everything out

$$\mathrm{E} = \exp\left\{ -\frac{1}{2} \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix} \right\} \tag{3.31}$$
$$= \exp\left\{ -\frac{1}{2} \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix}^T \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} (\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}_{22}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix} \right\} \tag{3.32}$$
$$= \exp\left\{ -\frac{1}{2} \begin{bmatrix} \boldsymbol{y}_1^T - \boldsymbol{y}_2^T(\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}) \\ \boldsymbol{y}_2 \end{bmatrix}^T \begin{bmatrix} (\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}_{22}^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{y}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{y}_2) \\ \boldsymbol{y}_2 \end{bmatrix} \right\} \tag{3.33}$$
$$= \exp\left\{ -\frac{1}{2} \begin{bmatrix} (\boldsymbol{y}_1^T - \boldsymbol{y}_2^T\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21})(\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} \\ \boldsymbol{y}_2^T\boldsymbol{\Sigma}_{22}^{-1} \end{bmatrix}^T \begin{bmatrix} \boldsymbol{y}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{y}_2) \\ \boldsymbol{y}_2 \end{bmatrix} \right\} \tag{3.34}$$
$$= \exp\left\{ -\frac{1}{2}(\boldsymbol{y}_1^T - \boldsymbol{y}_2^T\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21})(\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1}(\boldsymbol{y}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{y}_2) \right\} \times \tag{3.35}$$
$$\times \exp\left\{ -\frac{1}{2}\boldsymbol{y}_2^T\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{y}_2 \right\}$$

We can immediately see that the second term is a quadratic form in $\boldsymbol{x}_2$ and corresponds to $\mathcal{N}(\boldsymbol{x}_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22})$. Let us now consider the first term in isolation and move the terms around a little bit. We also make use of the fact that because $\boldsymbol{\Sigma}_{22}$ is a positive-definite matrix, its inverse is also symmetric, so $\boldsymbol{\Sigma}_{22}^{-1^T} = \boldsymbol{\Sigma}_{22}^{-1}$. We also know that $\boldsymbol{\Sigma}_{12}^T = \boldsymbol{\Sigma}_{21}$.

$$E_{1|2} = \exp\left\{ -\frac{1}{2}(\boldsymbol{y}_1^T - \boldsymbol{y}_2^T\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21})(\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1}(\boldsymbol{y}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{y}_2) \right\} \tag{3.36}$$
$$= \exp\left\{ -\frac{1}{2}(\boldsymbol{y}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{y}_2)^T(\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1}(\boldsymbol{y}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{y}_2) \right\} \tag{3.37}$$
$$= \exp\{ -\frac{1}{2}(\boldsymbol{x}_1 - \boldsymbol{\mu}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{x}_2 - \boldsymbol{\mu}_2)^T(\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} \tag{3.38}$$
$$(\boldsymbol{x}_1 - \boldsymbol{\mu}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{x}_2 - \boldsymbol{\mu}_2))\} \tag{3.39}$$

In equation 3.39 we again see a Gaussian density with parameters

$$\boldsymbol{\mu}_{1|2} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{x}_2 - \boldsymbol{\mu}_2) \tag{3.40}$$

$$\boldsymbol{\Sigma}_{1|2} = (\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}{\boldsymbol{\Sigma}_{22}}^{-1}\boldsymbol{\Sigma}_{21}. \tag{3.41}$$

This formula extremely important for the use of GPs as a probabilistic model in bayesian optimization. It will allow us to compute the exact parameters of the posterior $p(f|x)$ at any given point, and as a result compute the acquisition function.

marginal

## 3.4 Gaussian Processes

**Gaussian Process** is a stochastic process (a collection of random variables), such that every subset of those random variables has a multivariate Gaussian distribution. It is defined by a mean function $m(\boldsymbol{x})$ and a covariance function $\kappa(\boldsymbol{x}, \boldsymbol{x}')$. Formally, we write

$$p(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), \kappa(\boldsymbol{x}, \boldsymbol{x}')). \tag{3.42}$$

Any finite subset $\boldsymbol{x} = (x_1, \dots, x_n)$ is jointly Gaussian with mean vector $m(\boldsymbol{x})$ and covariance matrix $\boldsymbol{\Sigma}(\boldsymbol{x})$ where $\boldsymbol{\Sigma}(\boldsymbol{x})_{ij} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where $\kappa$ is any positive definite kernel function [Murphy and Bach, 2012].

The GP defines a prior distribution over functions $f$, which when combined with data $x$ can be converted into a posterior distribution over functions $p(f|x)$.

### 3.4.1 GP regression with noise-free observations

Consider the case when we are interested in predicting a function $f$ based on a few observations $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$, and we are interested in predicting the value of $y_\star$ at a new point $x_\star$.

Using the definition of a GP, we know that $y$ and $y_\star$ are jointly Gaussian. We also know, that these are the only points we are interested in. Even though the GP is a distribution over functions, that is over infintiely dimensional vectors, we only need to look at finitely many points and can ignore the rest. This is a crucial property of the GP and essentially makes everything we are about to do possible.

By assuming a GP, we get all of the properties of a multivariate Gaussian for free, including a closed form solution to the conditional and marginal distribution parameters. Let us now write the joint distribution of $\boldsymbol{y}$ and $\boldsymbol{y}_\star$ in a partitioned form

$$\begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{y}_\star \end{pmatrix} \sim \mathcal{N}\left( \begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_\star \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_\star \\ \mathbf{K}_\star^T & \mathbf{K}_{\star\star} \end{pmatrix} \right)$$

where $\mathbf{K} = \kappa(\mathbf{X}, \mathbf{X})$, $\mathbf{K}_\star = \kappa(\mathbf{X}, \mathbf{X}_\star)$ and $\mathbf{K}_{\star\star} = \kappa(\mathbf{X}_\star, \mathbf{X}_\star)$ [Williams and Rasmussen, 2006]. Note that $\boldsymbol{y}$ and $\boldsymbol{y}_\star$ can be either single points, or they can be vectors, as we might be interested in computing the posterior over multiple points at once given an existing dataset. Because this is just a multivariate Gaussian, we can make use of the conditioning formula and compute the posterior

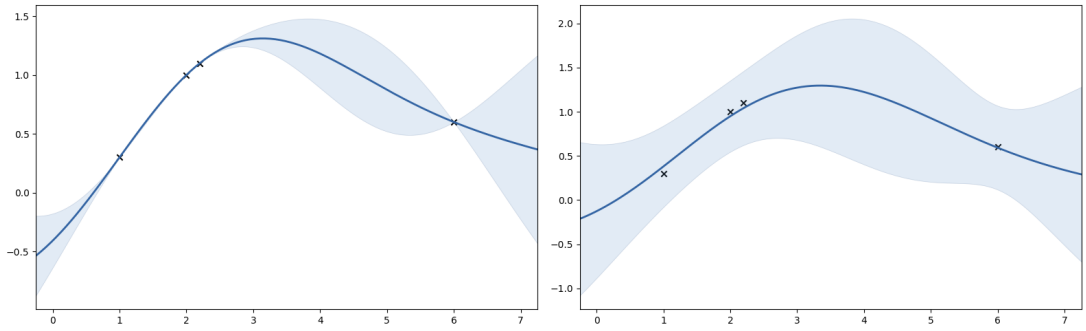

Figure 3.1: GP regression without noise on the left, and with a constant amount of noise added on the right.

$p(\boldsymbol{y}_\star \mid \mathbf{X}_\star, \mathbf{X}, \boldsymbol{y})$ exactly as

$$p(\boldsymbol{y}_\star \mid \mathbf{X}_\star, \mathbf{X}, \boldsymbol{y}) = \mathcal{N}(\boldsymbol{y}_\star \mid \boldsymbol{\mu}_\star, \boldsymbol{\Sigma}_\star) \tag{3.43}$$

$$\boldsymbol{\mu}_\star = \boldsymbol{\mu}(\mathbf{X}_\star) + \mathbf{K}_\star^T \mathbf{K}^{-1}(\boldsymbol{y} - \boldsymbol{\mu}(\mathbf{X})) \tag{3.44}$$

$$\boldsymbol{\Sigma}_\star = \mathbf{K}_{\star\star} - \mathbf{K}_\star^T \mathbf{K}^{-1} \mathbf{K}_\star. \tag{3.45}$$

where $\boldsymbol{\mu}_\star$ is the mean and $\boldsymbol{\Sigma}_\star$ the covariance of the multivariate Gaussian on $\boldsymbol{y}_\star$.

### 3.4.2 GP regression with noisy observations

Consider the case when $f$ is not a deterministic function, but rather a stochastic function which returns a noisy output $y$ given some fixed input $\boldsymbol{x}$ (meaning $f(\boldsymbol{x})$ is a random variable). GP regression is flexible enough to model Gaussian noise in the output directly. For now, let us consider the noise having a fixed variance $\sigma^2$.

In practice, the noise becomes a baseline for the variance of each point of the posterior, as the variance can never be lower than the noise. Since the variance is represented on the diagonal of the covariance matrix computed by the kernel function $\kappa$, we can model the noise directly by simply adding a diagonal matrix to the output of the kernel, that is

$$\operatorname{cov}(\boldsymbol{y}) = \kappa(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}.$$

See Figure 3.1 for a comparison of noise-less and noisy regression. It should also be noted that in principle nothing is preventing us from specifying a different noise value for each element of the diagonal. This could be useful if we had additional prior information about the function $f$. It could, for example, be an output of a measurement for which we know exactly the amount of noise for each $\boldsymbol{x}$.

### 3.4.3 Kernels

So far we have considered the covariance function $\kappa$ to be an arbitrary positive-definite kernel. Even though in theory there are no restrictions on what kernel we can choose, there are a few more popular choices that are commonly used.

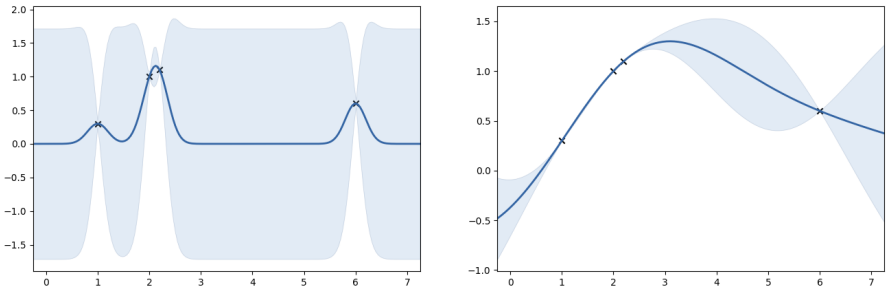

Figure 3.2: Lengthscale $l = 0.2$ on the left, and $l = 2$ on the right.

A prototypical example is the squared exponential (SE) kernel, also called the radial-basis function (RBF) kernel

$$\kappa(x_1, x_2) = \sigma_k^2 \exp\left\{-\frac{1}{2l}(x_1 - x_2)^2\right\}.$$

This kernel, among many others, falls under the category of stationary kernels. A **stationary kernel** is one which is shift-invariant, which means its value does not depend on the absolute values of $x_1$ and $x_2$, but only on their distance $d = |x_1 - x_2|$. We can thus write it as

$$\kappa(x_1, x_2) = \sigma_k^2 \exp\left\{-\frac{1}{2l}d^2\right\}.$$

The values $\sigma_k$ and $l$ are called the variance and lengthscale, and control the behavior of the kernel, where $\sigma_k$ changes the vertical scale of the function, and $l$ changes the horizontal scale. Changing the lengthscale essentially allows the kernel to re-normalize the data. If $\boldsymbol{x}$ is a vector we can define a lengthscale parameter $l_i$ for each of the components. This becomes very useful in the context of hyperparameters where each hyperparameter can have a very different scale, and the individual lengtscale per component allows the kernel to capture it.

Figure Figure 3.2 show how the behavior of the kernel changes based on the changed value of the lengthscale $l$. When the lengthscale is set too low the values of $y$ become essentially uncorrelated, leading to a function with many spikes. On the other hand, a larger value for the lengthscale yields a much smoother function.

A popular kernel in the context of Bayesian optimization is the Matérn kernel

$$\kappa(d) = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu}d}{l}\right)^{\nu} K_\nu\left(\frac{\sqrt{2\nu}d}{l}\right)$$

where $\nu$ and $l$ are the parameters and $K_\nu$ is a modified Bessel function [Williams and Rasmussen, 2006]. In practice, we will restrict ourselves to a half-integer variant of the matern kernel, that is when $\nu = p + 1/2$, where $p$ is a non-negative integer. In this case the Matérn kernel has a simplified form, specifically let us show for $\nu = 5/2$, which is popular in ML. The covariance function then simplifies to

$$\kappa(d) = \left(1 + \frac{\sqrt{5}d}{l} + \frac{5d^2}{3l^2}\right)\exp\left(-\frac{\sqrt{5}d}{l}\right)$$

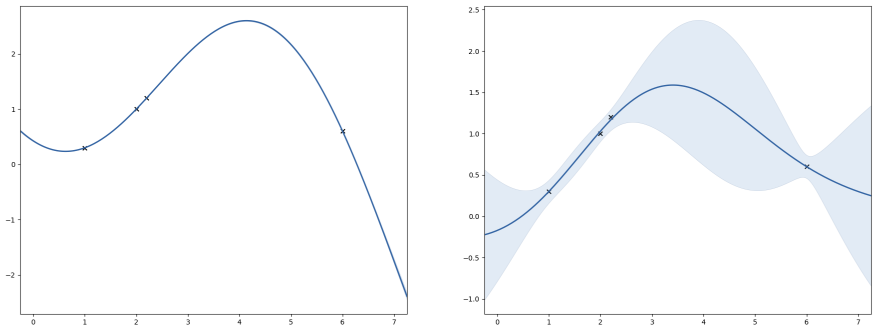

Figure 3.3: GP regression without optimizing kernel parameters on the left, and after optimizing using maximum likelihood on the right. Notice that the noise parameter is also optimized, as the regression model does not go directly through the data points, but instead considers the small variation as due to noise.

which ends causing the GP posterior to be 2 times differentiable [Williams and Rasmussen, 2006], as compared to the RBF kernel shown above, which is infinitely differentiable. As a side note, the Matérn kernel converges to the RBF kernel as $\nu \to \infty$.

In a general setting it could prove useful to examine many different kernel functions and use one that works best in a particular domain. As our case is Bayesian optimization, we will stick with the kernels shown above, as those are most often shown to perform well in literature, especially the 5/2 Matérn kernel [Snoek et al., 2012].

### 3.4.4 Optimizing GP hyperparameters

So far we have considered the noise, variance and lengthscale parameters to be fixed, but in this section we show how their value can be determined automatically form the data using maximum likelihood estimation.

Since the GP is a probabilistic model, we can ask it directly what is the likelihood of our data. Using the definition of a GP, we know that the likelihood of our data is a multivariate Gaussian, that is $p(\boldsymbol{y}|\mathbf{X}) = \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{K})$, giving us a log likelihood of

$$\log p(\boldsymbol{y}|\mathbf{X}) = -\frac{1}{2}\boldsymbol{y}\mathbf{K}^{-1}\boldsymbol{y} - \frac{1}{2}\log\det\mathbf{K} - \frac{N}{2}\log(2\pi).$$

We leave out the technical details (see Williams and Rasmussen [2006] for more details) including the gradient of the marginal likelihood with respect to the kernel parameters, as they do not provide any useful insights.

One important detail we want to stress out is that computing $K^{-1}$ takes $O(N^3)$, which puts a serious restriction on the size of the data we can fit with exact GP regression. This does not concern us in the context of hyperparamter optimization as we would always stay within low hundreds of evaluations anyway, but for other tasks it becomes a serious limitation. As a result many workarounds for approximate inference were developed [Williams and Rasmussen, 2006], which

again, we omit from this text, because they are not relevant for hyperparameter optimization.

Implementing the kernel parameter optimization is simple in practice. One can simply implement the marginal likelihood formula shown in Figure 3.4.4 and use a package for automatic differentiation with an optimizer like SGD or L-BFGS to optimize the parameters. Since the objective is non-linear, a common practice [GPy, since 2012] is to optimize with mutliple restarts. Figure 3.3 shows the effect of optimizing kernel parameters using maximum likelihood. More details on optimizing the kernel parameters can be found in Section 4.6.

neni oz-
naceno
jako
rovnice

# 4. Technical Details of BO

This chapter provides a more technical insight into Bayesian optimization. We begin by looking at the acquisition functions from a mathematical perspective, as they form the basis of Bayesian optimization. Next, we show how to extend Bayesian optimization to perform parallel evaluations, work with integer and discrete hyperparameters, and optimize parameters on a logarithmic scale. And finally, we explore the Bayesian optimization algorithm in detail, including some of its numerical properties and issues that can arise when implementing it.

The contents of this chapter are largely implementation details and are not necessary for one to use Bayesian optimization in practice. However, understanding the behavior of integer based hyperparameters, in addition to the overview in Section 2.2, can prove useful when deciding if a certain hyperparameter makes for a good candidate for automatic tuning using Bayesian optimization.

## 4.1 Acquisition functions

The acquisition function is a key component of Bayesian optimization. Together with the GP regression it allows us to balance the exploration-exploitation problem in search. The only limitation we need to impose on the acquisition function is tractability, and possibly continuity, as we will need to optimize it. The tractability requirement is mandatory, as without being able to compute the function we could hardly find its maximum. But the continuity requirement is useful as the acquisition function is often optimized using stochastic gradient optimizers such as L-BFGS.

An intuitive choice for an acquisition function is to maximize the probability of improving over our currently best achieved value, which is called the **probability of improvement**. This can be computed in closed form as

$$\mathrm{PI}(\boldsymbol{x}) = \Phi\left(\frac{\mu(\boldsymbol{x}) - y_{\max}}{\sigma(\boldsymbol{x})}\right)$$

where $y_{\max}$ is the maximum value achieved by sampling $f(\boldsymbol{x})$.

A natural extension is the **expected improvement** (EI) acquisition function which is simply the expected improvement over the currently achieved maximum. We define it as

$$\mathrm{EI}(x) = \mathbb{E}[\max(0, f(x) - y_{\max})].$$

At first it might seem that the expectation would be an intractable integral, but fortunately even this equation can be computed in closed form as

$$\mathrm{EI}(x) = \Delta(x) + \sigma(x)\varphi\left(\frac{\Delta(x)}{\sigma(x)}\right) - |\Delta(x)|\Phi\left(\frac{\Delta(x)}{\sigma(x)}\right)$$

where $\Delta(x) = \mu(x) - y_{\max}$. In practice, improvement shows better results than probability of improvement. For more examples of acquisition functions see Frazier [2018].

In both of these cases, the next sampling point would be chosen by maximizing the acquisition function, that is

$$x_{\text{next}} = \arg\max_x EI(x)$$

for the case of expected improvement. This can again be done by any stochastic optimizer, such as the commonly chosen L-BFGS with restarts.

## 4.2 Parallel Evaluations

In practice we might have the ability to evaluate $f(\boldsymbol{x})$ at multiple points in parallel, but the framework we have shown so far only allows for sequential optimization. In the previous section we've shown a few examples of the acquisition functions. A natural extension would be to not optimize with respect to a single $x_{\text{next}}$, but rather multiple points. In the context of EI this is called **parallel expected improvement**.

Unfortunately, there is no simple solution [Frazier, 2018]. A common solution is the so called **Constant Liar** approximation, which chooses $x_{i+1}$ assuming $x_i$ was already chosen, and has the corresponding value $y_i$ equal to a constant, often chosen to be the expected value of $f(x_i)$ under the GP posterior.

This allows us to trivially implement parallel evaluations by simply considering the $\mu$ prediction for unfinished evaluations as their $y$ value and consider them part of the dataset $\mathcal{D}$.

## 4.3 Integer Hyperparameters

GP regression by itself does not have the ability to model integer values in $X$ directly as some other models do (e.g. random forests Chapter 2). A common solution, used by [Group, 2014] and which we also implement in this thesis, is to consider all parameters to be real valued and only round them at the end.

In recently published work by Garrido-Merchán and Hernández-Lobato [2017] they show a more principled approach. The effect of rounding causes the model to see variation and relationships even among constant-valued regions. A possible downside is that the model could predict values different enough so that the acquisition function would obtain a maximum within a constant region which already has an existing sample, and thus wasting an evaluation. A proposed solution to this problem, as mentioned in the paper, is to round the appropriate values right before they are input into the kernel function, such as

$$\kappa'(x_1, x_2) = \kappa(T(x_1), T(x_2))$$

where $T(x)$ is an identity for real valued elements and a rounding function for integers.

Our implementation however does not use this approach, as our GP regression is handled by the GPy [since 2012] library, which did not support it at the time, and implementing it would mean overriding many of the existing kernels. We did instead handle the problem explicitly by detecting the pathological cases, as described in Chapter 5.

## 4.4 Logarithmic Scaling

When optimizing hyperparameters we might want to distinguish not only between real and integer valued ones, but also based on their scale. Optimizing the number of training epochs or layers are very well modelled by a linear scale, but a learning rate is better modelled with a logarithmic scale.

We provide a simple solution, which can work independently of Bayesian optimization, by simply transforming all of the appropriate value to logscale before inputting them into the model, and then transforming them back after we get a next sample $x$ proposal. As this approach is completely transparent from the point of the GP regression model, we could just as well perform any other arbitrary bijection.

## 4.5 Implementation Details of Bayesian Optimization

We will now show the algorithm for Bayesian optimization in greater detail as compared to algorithm 1. Let $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i \in 1:n\}$ denote a set of $n$ samples (evaluations) of the objective function $f$, that is $y_i = f(\mathbf{x}_i)$. Our goal is to pick the next $\mathbf{x}_{n+1}$ to maximize our chance of finding the optimum quickly, assuming that already enough points were evaluated for us to fit the GP. 2 shows one iteration of Bayesian optimization as it picks $\boldsymbol{x}_{n+1}$.

Let $\mathbf{X}$ be the matrix of all $\boldsymbol{x}_i$, and $\boldsymbol{y}$ be a column vector of all $y_i$
Maximize the kernel log-likelihood
$p(\mathbf{X}) = -\frac{1}{2}\boldsymbol{y}\mathbf{K}^{-1}\boldsymbol{y} - \frac{1}{2}\log\det\mathbf{K} - \frac{N}{2}\log(2\pi)$, where $\mathbf{K} = \kappa(\mathbf{X}, \mathbf{X})$, by tuning the kernel hyperparameters
Maximize the acquisition function $A(\mathbf{X}, \boldsymbol{y}, \mathcal{GP})$ as a function of the GP using the kernel hyperparameters obtained in the previous step.
Sample $y_{n+1} = f(\boldsymbol{x})$ where $\boldsymbol{x} = \arg\max A(\mathbf{X}, \boldsymbol{y}, \mathcal{GP})$.
Add $y_{n+1}$ to the dataset as $\mathcal{D}_{i+1} = \mathcal{D}_i \cup (\boldsymbol{x}_{i+1}, y_{i+1})$.

**Algorithm 2:** Bayesian Optimization with implementation details.

While the above shown algorithm is enough to explain how Bayesian optimization works, there are a few cases where numerical issues can arise, and we point them out next.

Firstly, the quadratic form $\boldsymbol{y}\mathbf{K}^{-1}\boldsymbol{y}$ does not need to be computed using a matrix inverse procedure, which can be numerically unstable and requires more intermediate computation (see Krishnamoorthy and Menon [2013] for details). In general, consider the equation $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. We can instead write $\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b}$ for an invertible matrix $\mathbf{A}$. A naive solution would use a procedure for an inverse and then multiply to compute $\mathsf{inv}(\mathbf{A})\boldsymbol{b}$, but because we do not actually need the inverse itself, but rather the multiplication, we can solve for $\mathbf{A}^{-1}\boldsymbol{b}$ directly using a `solve` procedure [van der Walt et al., 2011].

Because the $\mathbf{K}$ matrix is a covariance matrix we also know it is positive definite. This allows us to take the computation one step further and compute its Cholesky factorization $\mathbf{L} = \mathsf{chol}(\mathbf{K})$, and then use a procedure for `solve` directly on the factorized matrix $\mathbf{L}$ (in TensorFlow available as `tf.linalg.cholesky_solve`).

Having just computed the Cholesky factorization we can re-use it in the second expression of the marginal log-likelihood, that is the $\log\det\mathbf{K}$, since

$$\log\det\mathbf{K} = \log\det(\mathbf{L}^T\mathbf{L}) = \log(\det(\mathbf{L}^T)\cdot\det(\mathbf{L})) = \tag{4.1}$$

$$= 2\cdot\log\det\mathbf{L} = 2\cdot\log\prod\operatorname{diag}\mathbf{L} = 2\cdot\sum\log\operatorname{diag}.\mathbf{L} \tag{4.2}$$

This allows us to compute the determinant, which is usually $O(n^3)$, in just linear time, because we're re-using the work already being done in the Cholesky factorization used in the previous step.

Another interesting note, discovered by our initial custom implementation, and confirmed in the implementation of the GPy [GPy, since 2012] library, is that when computing a Cholesky factorization of a covariance matrix computed on real world data, it will often fail for numerical reasons, and requires additional noise to be added to the diagonal to improve stability. What GPy does internally is to iteratively increase the amount of noise, up until some threshold, to make sure the factorization succeeds without any problems, while not adding excessive noise when not needed.

Lastly, the optimization procedures themselves used for maximizing the kernel log-likelihood and acquisition function can be sensitive. Our initial implementation in SciPy and TensorFlow showed very different restarts based on the type of optimizer (SGD, Adam, L-BFGS), and its meta-parameters, such as the number of restarts, stop tolerance criterion, learning rates, etc. These problems, along with the many numerical issues encountered along the way, contributed to the choice of using GPy for the final implementation. We outline a few more reasons in Section 5.2.

## 4.6 Priors on Kernel Parameters

One of the benefits of the GP regression model is that it is very flexible. Unlike parametric, such as linear regression, it can fit arbitrarily complex curves through the data. This flexibility is dictated by the choice of a kernel function, which determines individual covariances between every pair of points. As we have shown earlier (see Subsection 3.4.3) the kernels themselves have hyperparameters which greatly affect the behavior of the model. Even though it is theoretically possible to set the kernel parameters by hand, as it could be done if we knew some properties of the objective function, we cannot take this approach when fitting arbitrary samples from the loss landscape of a neural network. Instead we take a principled approach using statistics and optimize the kernel parameters to maximize the likelihood of the data, specifically using maximum likelihood estimation (MLE).

Unfortunately, because MLE is a point estimate, it is prone to overfitting. In our particular case of using a flexible non-parametric model we can run into much severe case of overfitting. In theory, some argue (see [Williams and Rasmussen, 2006]) that when the likelihood function is multi-modal it is usually because there are multiple interpretations to the data, and each mode maps to an intuitive interpretation. We have not found this to be true on more complex datasets, as those sampled from the objective function when using neural networks, and very often the MLE ends up choosing a model with extremely high capacity, as shown in Figure 4.1.

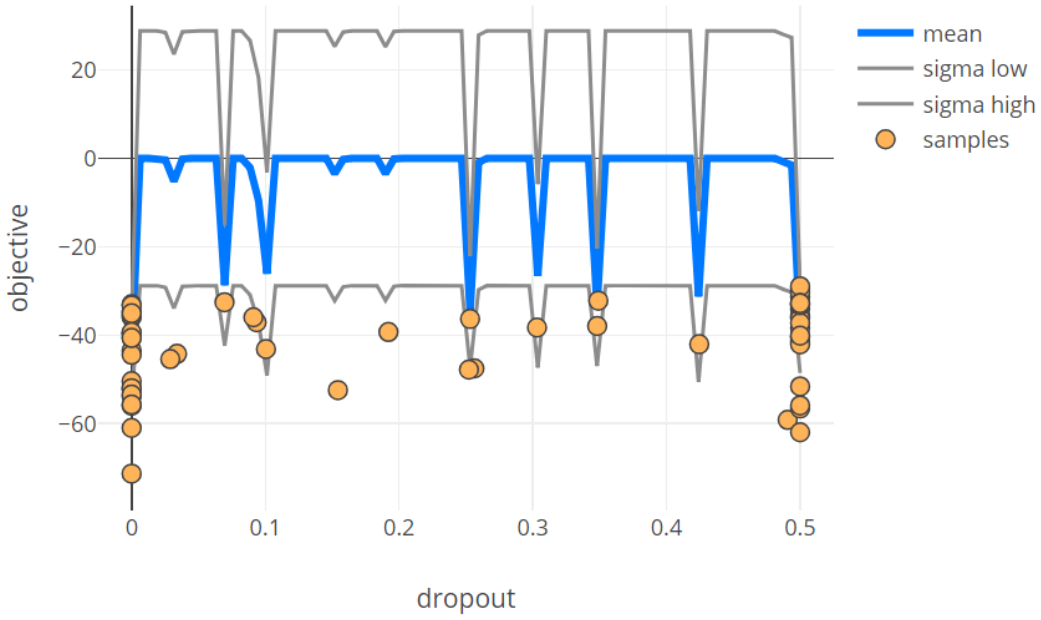


Figure 4.1: GP regression with a very low value for the lengthscale kernel parameter. Since the scale parameter is low, the $x$ axis ends up being *stretched out*, and each data point becomes independent, that is their covariance is low. As a result, the model is free to try to fit almost every data point independently of the others, and its predictions become uninformative.

A common solution to the overfitting problem of MLE is to introduce a prior distribution $p(w)$ on the parameters $p(w)$, and then instead of maximizing the likelihood $p(x|w)$ we would maximize the posterior distribution $p(w|x) \propto p(x|w)p(w)$. This procedure is commonly called the maximum a posteriori (MAP) estimate, and is commonly employed in machine learning models as a way of reducing overfitting. For example, in the case of linear regression, the standard least squares solution corresponds to maximizing the likelihood of the data using MLE, or equivalently minimizing the mean squared error between the predictions and the labels. A probabilistic extension of linear regression which introduces a Gaussian prior on the weights is called a Ridge regression, and corresponds to computing the MAP estimate of the parameters, or equivalently minimizing the mean squared error with an additional weight decay term.

In the ideal case, we would either marginalize over the parameters, or take a fully Bayesian approach and compute the posterior $p(w|x)$. Unfortunately, in many cases including ours, this becomes intractable due to the normalization constant given by the evidence term $p(x)$. The MAP estimate is a practical compromise between the frequentist point estimate using MLE, and a fully Bayesian treatment. Because we only need to compute the $\arg\max$ of the posterior we can optimize it without computing the normalization term $p(x)$, as $\arg\max$ is invariant to scaling by a constant. The MAP estimate is then simply computed as

$$\arg\max_{w} p(x|w)p(w),$$

where in our case $w$ represents the kernel parameters, as well as the constant for Gaussian noise in each sample.

As with any Bayesian method, the choice of a prior is of great importance. If we were to choose a uniform prior, computing the MAP estimate would be equivalent to computing the MLE with constrained optimization on the support of the uniform distribution. In practice we could take a conservative step and choose a non-informative prior, which would still act as regularization, and possibly help with overfitting as compared to computing a bare MLE estimate. An example of such prior is the $\mathsf{Gamma}(1.0, 0.001)$ distribution shown in Figure 4.2, as its support is positive real numbers, and both the variance and lengthscale of the kernel are also defined only for positive real numbers.

In our experiments, as described in Chapter 6, we show a few cases where the choice of an uninformative prior leads to poor model behavior. The common issue is when the model chooses a small lengthscale for hyperparameters with high range of values, as shown previously in Figure 4.1. Because the process of Bayesian optimization is built on top of the regression model it can become problematic when the model sees most of the search space as constant regions with only a few peaks at the sampled data points.

A possible solution to this problem is to abandon the idea of non-informative priors and instead take the approach often called Empirical Bayes [Murphy and Bach, 2012], where the parameters of the prior distribution are estimated from the data themselves. Choosing between a non-informative prior and estimating the parameters from data is a principal problem that does not have a definitive answer.

As our motivations are largely driven by practical applications rather than theoretical purity we do estimate the prior parameters from data in some visualizations (see Section 5.5) to avoid pathological cases and provide more useful user experience. We also show some comparisons between the non-informative priors and the Empirical Bayes approach in the GP regression model used when computing the acquisition function, as compared to only in visualizations, in Subsection 6.1.1. Unfortunately as some experiments of our experiments were very compute intensive (some over a thousand GPU-hours) we do not provide full ablation analysis.

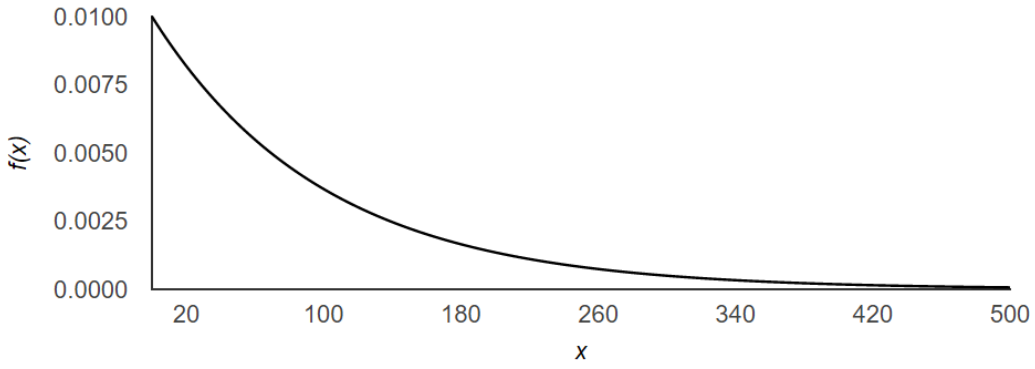


Figure 4.2: An example of a prior distribution defined as $\mathsf{Gamma}(1.0, 0.001)$ with support over positive real numbers.

# 5. Software

This chapter describes the implementation part of this thesis. While we do not provide any theoretical extensions to Bayesian optimization, we instead provide a modular and fully working implementation which was tested on multiple experiments. The implementation is provided as a Python (van Rossum [1995]) package called `bopt` (short for Bayesian optimization).

The main features of the package are:

- A robust implementation of Bayesian optimization.
- Flexible experiment configuration with random search and GP backends.
- Parallel execution of evaluations, both on the local machine and on a cluster.
- Robust error handling with duplicate/similar sample detection.
- Command line interface for controlling the experiment evaluations, including running a manual evaluation with user specified hyperparameters.
- Simple filesystem based storage with user-readable and editable serialization format based on YAML.
- Web based visualizations of the whole optimization process, including 1D and 2D slices and marginal plots at all points during the evaluation.
- Ability to add manual samples either from another, already executed, experiment. Or simply by running the appropriate command with user-provided hyperparameter values.

## 5.1 Architecture

In this section we will explore the high level architecture of `bopt` . Everything is structured around a central class `Experiment`, which represents a single objective function together with a configuration of its hyperparameters, and configuration for the Bayesian optimization itself. The `Experiment` can contain multiple `Samples`, where each sample represents a single evaluation of the objective function.

We assume the function being optimized can be evaluated by running a script file. The hyperparameters will be passed as command line arguments (see Section 5.4), and the output will be parsed from the standard output using a regular expression provided by the user. This provides the user with maximum flexibility with regards how the actual function is being executed, as `bopt` will simply spawn the process, pass the command line arguments, and then wait for it to terminate to collect the output and parse the result. If the result is not found in the output, or the process exists with an exit code greater than 0 it will mark the evaluation as failed. We do not put any restrictions on the type of script the user might want to provide. It is solely at the discretion of the `Runner` (see Subsection 5.1.2) to figure out how to run the provided file.

Each `bopt` experiment is located in its own directory on the filesystem (called the **`meta_dir`**), where it stores all of the information in a single `meta.yml` file, along with output files for each job. This makes it easy for the user to manually inspect and edit if needed, or even backup when performing more complicated operations, such as deleting specific samples, or manually adding samples from a different experiment. Since Bayesian optimization is memory-less, that is always starting from scratch, the user can easily combine evaluations from multiple different experiments by hand, or even delete samples which were created by accident, such as when using manual evaluations.

### 5.1.1 Samples, Result Collection and Locking

Each evaluation of the objective function is split into two parts. One being the `Sample`, which contains the specific hyperparameter values for $x$, kernel parameters of the GP model which was used to compute the sample, a posterior prediction of its mean and variance, and then the second part, which is an optional `Job` instance, which represents the actual running evaluation. The `Job` simply wraps the running process

Each `Sample` can also have an optional `Job`, which only wraps the process ID with runner-specific information on how to get the status of the running job, kill it, etc. The `Sample` only represents a snapshot at one point in time. Every time a `bopt` command is executed, or whenever a new state is required, a *result collection procedure* will be called, which checks the status of all running jobs, and updates their respective samples with any results or failure information. This is done mainly to avoid race conditions, as the evaluations themselves are running asynchronously from the main program flow in `bopt` . Apart from the collection procedure and a few exceptions, such as starting a new job, the main data structure is considered read only.

It is also worth mentioning, that since multiple instances of `bopt` could be running at any given time, we have employed a file locking mechanism which creates a `.lockfile` file in the experiment directory, which any other `bopt` instance would detect and wait on release. This allows the user to use the command line utilities while an experiment is running without worrying about data corruption.

### 5.1.2 Runners

Training neural networks is a computationally intensive task, and tuning hyperparameters makes it an order of magnitude more expensive. As a result, running some experiment on a local computer might not be an option for the user. The package was designed with different evaluation environments in mind and provides a flexible concept of a `Runner` class, which abstracts away the process of starting a new evaluation, that is figuring out how and where to run the script file representing the objective function.

We provide two different runners out of the box:

**Local** runs the process on the same machine as `bopt` .

**SGE** submits a job to the Son of Grid Engine [sge].

All runners support job parallelism using the Constant Liar approximation (see Section 4.2), which is part of the reason why each `Sample` stores a mean prediction. This value is being used as $y$ whenever a new evaluation point needs to be chosen during parallel evaluations.

When a job is started, its stdout and stderr will be redirected to a file within an `output` directory in the `meta_dir`. The file will be named with a process ID (PID) in case of a *local* job, and with a job ID in case of an *SGE* job. In case of the local runner we have to employ a minor trick, because when `popen` is being called with stdout redirection it already requires a file handle, but at that time the process ID is unknown. We work around this by creating a temporary file, redirecting the output to that, and then after `popen` returns a PID we rename the opened temporary file to a new name with the PID. Since UNIX systems can handle renaming of open files this workaround causes no issues, but the behavior on Microsoft Windows is unclear. If the user needs to support Microsoft Windows they might have to provide their own runner which does not use PIDs, but rather generates the ID based on some other process which makes the ID available before the child process is spawned. The SGE runner does not run into this issue, because we only specify the filename as a parameter to `qsub`, which then handles the process scheduling, creation, redirection, and names the output file accordingly.

To implement a custom runner, the user only needs to subclass two classes, namely the `Job` and the `Runner`. The `Job` class needs to mainly implement an `is_finished` method (among a few other ones described in the abstract interface), which simply returns the status of the evaluation based on the implementation details specified in the runner. To implement the `Runner` the user needs to provide a `start` method, which accepts the values of all hyperparameters, and returns a new `Job`, which will later be evaluated. An additional requirement is that the `Job` needs to store its result in an appropriately named file, specifically `job.oID` in the `outputs` directory. This was done mainly to avoid some issues with process IDs as described earlier in this section.

Lastly, it is worth mentioning that `bopt` supports the UNIX `time` command for measuring the runtime of an evaluation. Because we allow running jobs on an SGE cluster we can not simply take the start time and end time and subtract them to get the total run time, because the job can sit in a queue for an arbitrary amount of time.

## 5.2 GPy

Our library of choice for GP regression is the GPy [since 2012] library, as it is the most stable and robust package available, and is still being actively developed. We're mainly interested in the `GPy.models.GPRegression` class which implements the regression model itself, and the `GPy.kern` module, which implements the kernel functions. We initially used our own custom implementation of GP regression (partly shown in Section 4.5) in TensorFlow [Abadi et al., 2015] and SciPy [Jones et al., 2001–], but despite the seemingly short and simple numeric algorithm for computing the regression, getting all the details right proved to be an exceedingly difficult task. Even simple numerical methods like Cholesky decomposition are often wrapped with layers of numerical stability tricks that

one does not get out of the box with libraries like NumPy [van der Walt et al., 2011].

For these reasons we ended up switching to the GPy library, which apart from a numerically stable implementation provides many additional benefits. Being built upon a general purpose parameter optimization library par, GPy allows the user to both put arbitrary constraints on each of the parameters, as well a specify a prior distribution which is then used when optimizing the kernel marginal likelihood.

## 5.3 Random Search

The core optimization loop has the ability to generate samples randomly, which is useful for two reasons. It allows us to create a comparative baseline where all samples are generated using random search, so that we can measure the benefits and improvement of Bayesian optimization. But it also serves as a way of bootstrapping the Bayesian optimization driven search. When choosing an initial first point of evaluation, we don't have any data to fit the model to. We could optimize the acquisition function on the prior, but since our prior has zero mean, it would simply be a uniform distribution. What we do instead is sample each hyperparameter randomly, until we have enough points to fit the probabilistic model.

The number of random samples can increase if we are using parallel evaluations, simply because if we need to start $N$ jobs at the same time, with no prior data, we have no way of even calculating a mean prediction. As a result, we always start the first $N$ evaluations using random search.

## 5.4 Command Line Interface

Since larger experiments will most likely always be run on a compute cluster we opted for a command line interface which can be easily used over an SSH connection, and is very flexible. The available subcommands of `bopt` are:

**init** Creates a new experiment with a given script, configuration of hyperparameters and runner options.

**exp** Prints out the current status of a given experiment, showing metadata on all the evaluations.

**web** Starts the web interface for visualizing the evaluations.

**run** Starts a run loop which tracks how many jobs are currently running, starting new as needed to fulfill the parallelism requirements, and collects the results as needed.

**run-single** Runs a single evaluation, regardless of how many are currently running.

**manual-run** Runs a single evaluation with hyperparameters provided by the user. This does not use Bayesian optimization and simply serves as an interface to manually start tasks as needed.

**suggest** Prints a suggestion for the next evaluation without running it, as well as an already formatted command for `manual-run` so that the user can inspect the hyperparameter values and run the command immediately if they see fit.

**debug** Starts a Python debugger with the given experiment loaded in, which can be useful both for diagnosing issues as well as exploring the internal data structures.

**clean** Kills all running jobs and removes all evaluations from the experiment, while keeping the initialization metadata. This command is basically just a shortcut for re-starting an experiment from scratch.

All commands are executed as `bopt COMMAND` and support the conventional `--help` command line arugment, which prints out all of the available options, as well as their descriptions.

### 5.4.1 Meta Directory, Data Corruption and `-C`

When an experiment is initialized all of its information will be stored in its own directory. This includes both the meta information about hyperparameters, run configurations and evaluations, as well as the job outputs themselves.

The `bopt` command was designed such that it will always try to acquire an exclusive lock on the directory using a `.lockfile`, which is to prevent any race conditions and possible data corruption from running multiple instances of `bopt` at the same time. Such scenario could easily occur when the user would run a long running `bopt run` command, while also exploring the results, and possibly starting a few more evaluations manually using `bopt run-single`, `bopt manual-run`, or even another `bopt run`. Because of the locking behavior, it is completely safe to run as many instances of `bopt` as are needed, and the user does not need to concern themselves with causing any data corruption via the command line interface. We also make sure to always serialize the data into a new file, and then atomically move over the existing one, in order to minimize possible data corruption when the `bopt` process is killed.

It is important to note that `bopt` was designed with manual user intervention in mind. As such, the `meta.yml` file, which contains all of the experiment information, was created to be easily human editable. But because `bopt` does not use the UNIX `flock` mechanism (as editors do not obey it), the user has to be wary of editing the file by hand while other instances of `bopt` are running. Because the `meta.yml` file is overwritten atomically, the user can even edit the file while `bopt` is running, but they have to make sure to save at the appropriate time, e.g. not to discard the changes that were just written after the file was loaded in the editor. This problem is unlikely to occur in editors like Vim, which will notify the user of the file being changed after it was read, but it is still does not prevent the user to overwrite it. If there are no existing `bopt` processes running, it is completely safe to alter the `meta.yml` file.

All of the `bopt` commands will also accept a `-C` command line argument, which specifies a directory to `cd` into before any of the main code is executed (similarly as a `Makefile` would behave). This behavior allows `bopt` to always assume it is

being executed from within the `meta_dir` and simplify many possible problems with storing paths in the config files. While this behavior is unlikely to affect the user in a negative way, it is still useful to know the semantics of the program.

We now explore two of the most important commands in more detail, `bopt init` and `bopt run`.

### 5.4.2 `bopt init`

Initializing experiments is an important feature and as such the command line interface has been streamlined to allow the user to input the arguments without complicated config files. An example of a common `bopt init` call could look something like the following:

```
bopt init \
  --param "batch_size:int:4:128" \
  --param "gamma:logscale_float:0.5:1.0" \
  --param "lr:logscale_float:1e-6:1e-1" \
  --param "dropout:float:0.1:0.6"
  -C experiments/reinforce \
  --runner sge \
  --ard=1 --gamma-prior=1 \
  --gamma-a=1.0 --gamma-b=0.001 \
  $PWD/reinforce.sh
```

Let us now go over the arguments one by one. The first four arguments specify four different hyperaparameters, namely batch_size, gamma, lr and dropout, each with a different type and range. The general format is `NAME:TYPE:MIN:MAX` where `NAME` can be an arbitrary name, `TYPE` can be one of `int`, `float`, `logscale_int`, `logscale_float`, and `discrete`, and `MIN:MAX` are simply the bounds of the hyperparameter. If the type of `discrete` is specified, instead of providing the bounds the user would provide a colon separated list of possible values, which would then be encoded as ordinal integers. An example of such discrete hyperparameter could be an activation function defined as `activation:discrete:tanh:relu:sigmoid`. However, as mentioned in Section 2.2, we do not recommend using `bopt` for architecture search, which discourages from most uses of the `discrete` type.

The next argument `-C experiments/reinforce` simply defines the `meta_dir` where the experiment data will be stored. Next we define the runner type, which can be one of `local` and `sge`.

After the runner is defined, we configure the GP regression itself, in this case by specifying the `ard` flag on, which uses a separate lengthscale parameter for each component of $x$ (more details can be found in the `?` documentation). Next we specify that we want to use a Gamma prior on the hyperaparameters, and its shape and scale parameters. A complete list of all flags for configuring the GP regression, acquisition function, kernel and the optimizer can be found using the help flag as `bopt init --help`.

Lastly, we provide `bopt` with the script to run, in this case it is `reinforce.sh`, which encapsulates our objective function. We also specify it as an absolute path using the `PWD` environment variable, but this is shown mainly as an interesting

trick that can be useful if the `PATH` is not configured in the environment where the runner will execute the job.

After the command exists, it will create a directory `experiments/reinforce` with a `meta.yml` file inside. The following listing shows the contents of the file

```
gp_config: !!python/object:bopt.gp_config.GPConfig
  acq_n_restarts: 25
  acq_xi: 0.001
  acquisition_fn: ei
  ard: true
  gamma_a: 1.0
  gamma_b: 0.001
  gamma_prior: true
  kernel: Mat52
  num_optimize_restarts: 10
  random_search_only: false
hyperparameters:
  batch_size:
    high: 128
    low: 4
    type: int
  gamma:
    high: 1.0
    low: 0.5
    type: logscale_float
  hidden_layer:
    high: 128
    low: 2
    type: int
  learning_rate:
    high: 0.1
    low: 1.0e-06
    type: logscale_float
result_regex: RESULT=(.*)
runner:
  arguments: []
  manual_arg_fnames: []
  qsub_arguments: []
  runner_type: sge_runner
  script_path: ./reinforce.sh
samples: []
```

Apart from the command line arguments we have provided it should be noted that a few unspecified values were filled in with the defaults. For example, the kernel function was chosen to be the default Matérn 5/2 kernel, which was shown [Snoek et al., 2012] to perform the best on many hyperparameter tuning tasks. We have tried to make the optimization procedure as configurable as possible in case the user has any additional prior knowledge which might help them optimize better.

In general, there are no significant requirements on the script, as we execute it as a subprocess. We only require that it takes the hyperparameter values as command line arguments in the format of `--NAME=VALUE`, and outputs the objective function on its standard output. By default, we will parse the standard output with a regex `RESULT=(.*)`, but the user is free to modify this in the `meta.yml` file however they see fit. If the user wishes to run an existing software which does not accept command line arguments in this form, they have to wrap the program in a script which will pre-process the arguments given by `bopt` , and pass them through in the format they require. This approach allows for maximum flexibility without having to spend large amounts of effort on building a general argument passing system. In our experiments with existing software we only found a few cases where minor argument processing was necessary.

### 5.4.3 `bopt run`

After the experiment was initialized the user can start running evaluations. Since everything is already configured in the `meta.yml` file the user simply needs to run `bopt run -C experiments/reinforce`, or `cd` into the directory and just run `bopt run`. Both of these variants are equivalent.

By default, this would run 20 evaluations in total with no parallelism. The number of evaluations can be controlled with the `--n_iter` switch, while the number of jobs running in parallel is controlled by the `--n_parallel` switch.

The way `bopt run` tracks the number of running jobs is by checking the `meta.yml` file for the IDs, and then checking the job status and counting how many are running. This allows it to respect the number of jobs running prior to the execution of `bopt run`. If the user first was to start say 5 evaluations by hand (e.g. using `bopt run-single`) and only then run `bopt run --n_parallel=5` while the first 5 jobs were still running, the instance of `bopt run` would correctly identify the running jobs and wait for some of them to finish before launching new ones.

## 5.5 Visualizations

Given the complex nature of tuning hyperparameters, one might be tempted to simply run grid search and examine the results. Ignoring the computational aspects for a moment, let us focus on the manual inspection of the results. As the number of hyperparameters grows beyond to 5 – 10 it becomes very difficult to infer relationships between hyperparameters from a flat list of evaluations. Figure 5.5 shows an example of a table with 6 different hyperparameters. To model a 6-dimensional space one needs to run at least 15 – 20 evaluations to get enough information to infer relationships between the dimensions. But as the number of evaluations grow, it becomes increasingly difficult to look at tabular data and infer relationships from them.

We provide a practical solution of plotting 1D and 2D marginal GP fits for all hyperparameters and their combinations. In Figure 5.5 we show the relationship between two of the hyperparameters as measured in one of our experiments (more details in Chapter 6). We can also show each 1D marginal in order to visualize how each hyperparameter affects the fitness irrespective of the others, as shown

| Sample | Hyperparameters | Result | Model params | Created at | Done at | Run time | Comment |
|---|---|---|---|---|---|---|---|
| 5413233 OK | 93 0.9 0.5 0.1 308 3 0.3 | 97.44 | gpy 0.1 141 0.1 0.6 18 882 5 17 9.0 | 04.10 21:33 | 04.10 23:31 | 1:57 | None |
| 5411451 OK | 93 1.0 0.5 0.1 491 3 0.3 | 97.38 | gpy 0.3 122 342 522 882 1000 0.9 635 0.9 | 04.10 18:14 | 04.10 20:53 | 2:38 | None |
| 5409316 OK | 54 1.0 0.4 0.1 305 3 0.2 | 97.37 | gpy 0.0 2 2 2 2 2 2 2 1.1 | 04.10 04:50 | 04.10 07:02 | 2:12 | None |
| 5409911 OK | 86 0.9 0.3 0.1 453 3 0.1 | 97.37 | gpy 0.1 24 2 2 3 17 0.9 2 1.0 | 04.10 11:40 | 04.10 17:26 | 5:46 | None |
| 5409428 OK | 78 1.0 0.5 0.2 300 3 0.1 | 97.36 | gpy 0.0 2 2 2 2 2 2 2 1.1 | 04.10 06:39 | 04.10 08:17 | 1:37 | None |
| 5410841 OK | 92 1.0 0.4 0.1 486 3 0.2 | 97.35 | gpy 0.3 95 264 221 615 1000 0.8 873 0.8 | 04.10 18:02 | 04.10 23:41 | 5:39 | None |
| 5413838 OK | 76 0.9 0.6 0.0 467 3 0.0 | 97.35 | gpy 0.1 225 0.2 0.9 19 1490 13 65 24.3 | 04.11 04:42 | 04.11 06:43 | 2:1 | None |
| 5409290 OK | 74 1.0 0.5 0.1 488 3 0.4 | 97.34 | gpy 0.0 2 2 2 2 2 2 2 1.1 | 04.10 04:17 | 04.10 06:11 | 1:53 | None |
| 5413234 OK | 90 1.0 0.5 0.0 427 3 0.4 | 97.34 | gpy 0.1 146 0.1 0.6 18 896 5 16 9.1 | 04.10 21:34 | 04.10 23:32 | 1:58 | None |
| 5413966 OK | 73 1.0 0.4 0.0 371 3 0.3 | 97.34 | gpy 0.1 173 0.2 0.7 16 1752 9 54 14.1 | 04.11 07:35 | 04.11 10:02 | 2:27 | None |
| 5409074 OK | 85 1.0 0.5 0.0 473 3 0.1 | 97.32 | random search | 04.10 01:27 | 04.10 04:11 | 2:44 | None |
| 5413217 OK | 95 0.9 0.5 0.2 284 3 0.2 | 97.32 | gpy 0.0 0.0 7 486 79 1000 0.0 486 1.0 | 04.10 21:26 | 04.10 23:41 | 2:15 | None |
| 5413777 OK | 45 1.0 0.5 0.0 294 3 0.3 | 97.32 | gpy 0.1 200 0.2 0.9 18 1526 11 38 20.2 | 04.11 03:18 | 04.11 04:58 | 1:40 | None |
| 5413261 OK | 69 1.0 0.5 0.0 349 3 0.1 | 97.31 | gpy 0.4 1000 770 311 667 1000 2 464 1.5 | 04.10 22:00 | 04.11 03:14 | 5:13 | None |

Figure 5.1: A table showing the results of multiple objective function evaluations.

in Figure 5.5. The 1D figures can also plot the acquisition function, which can also serve as a useful debugging tool, e.g. to diagnose possible overfitting of the GP. Lastly, we also allow the user to view these visualizations at any point in time during the optimization process, as shown in Figure 5.4. This allows to view the GP regression as more evaluations were created.

The benefit of a GP regression model is that the marginal distribution on any combination if the hyperparameters simply follows the marginalization property (see Section 3.3.1), which says we can only look at the mean and covariance of the parameters we are interested in, as all of the other ones get marginalized out, that is $p(\boldsymbol{x}_2) \sim \mathcal{N}(\boldsymbol{x}_2 \,|\, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22})$ where $p(\boldsymbol{x}_2) = \int p(\boldsymbol{x}_1, \boldsymbol{x}_2)\, d\boldsymbol{x}_1$ and a partitioned matrix $\boldsymbol{\Sigma}$ as described in Chapter 3. Using this property, we can compute the 1D marginal projection by fitting a model to the coordinate corresponding to the hyperparameter of interest. As the goal of the marginal plots is to quickly see the trends in the data, we employ the Empirical Bayes approach described in Section 4.6 to bias the prior distribution on kernel parameters towards smoother kernel functions, to avoid pathological cases of overfitting as shown in Figure 4.1.

Similarly, we might be interested in looking at slices through the GP regression model, that is fixing a value of some hyperparameters, and examining how the objective changes when interpolating through the remaining ones. This is again simple to achieve computationally when the model is a GP, because by slicing we are simply conditioning on the values of some elements of $\boldsymbol{x}$ while leaving the others free. Using the conditioning formula shown in Equation 3.41 we can compute the posterior parameters in closed form, and then simply plot the predicted mean and variance. We don't show this case in the figures since the plots look exactly the same as the marginal ones, except of course for the specific values. The user can simply toggle between the two modes.

The kernel parameters for the conditioned GP are set to the exact values computed when the sample was chosen for evaluation. This way we can explore the progress of the optimization process through time and see what the model

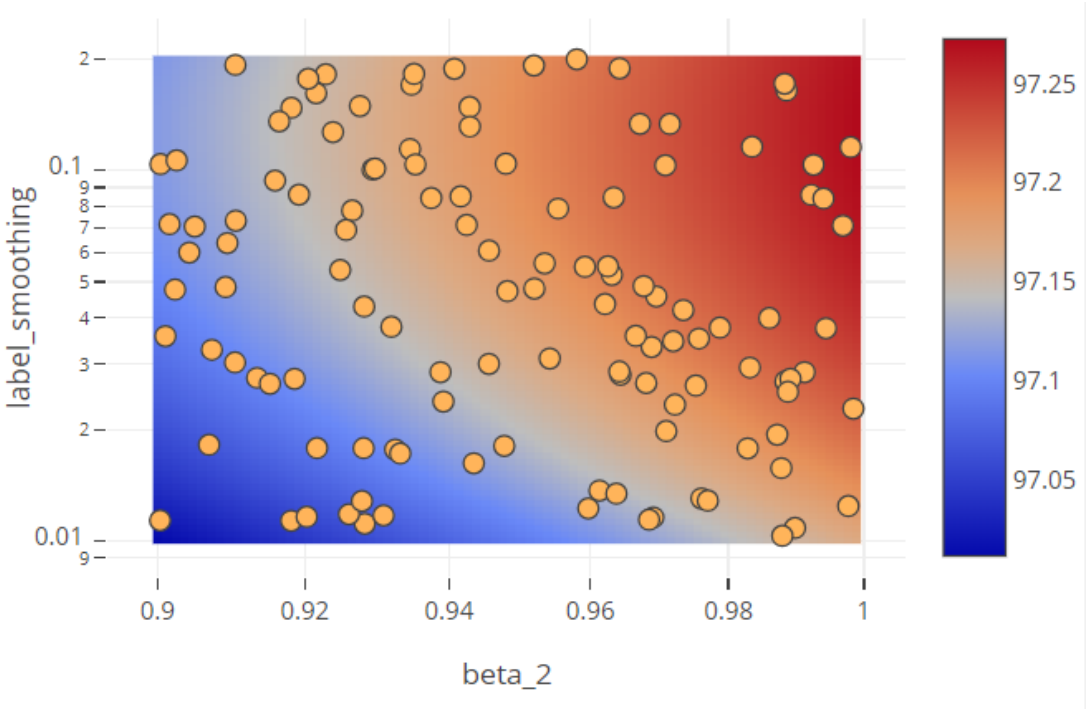


Figure 5.2: 2D marginal plot showing the dependence between $\beta_2$ and *label smoothing* in one of our experiments when training a larger tagger and lemmatizer network on a Czech treebank.

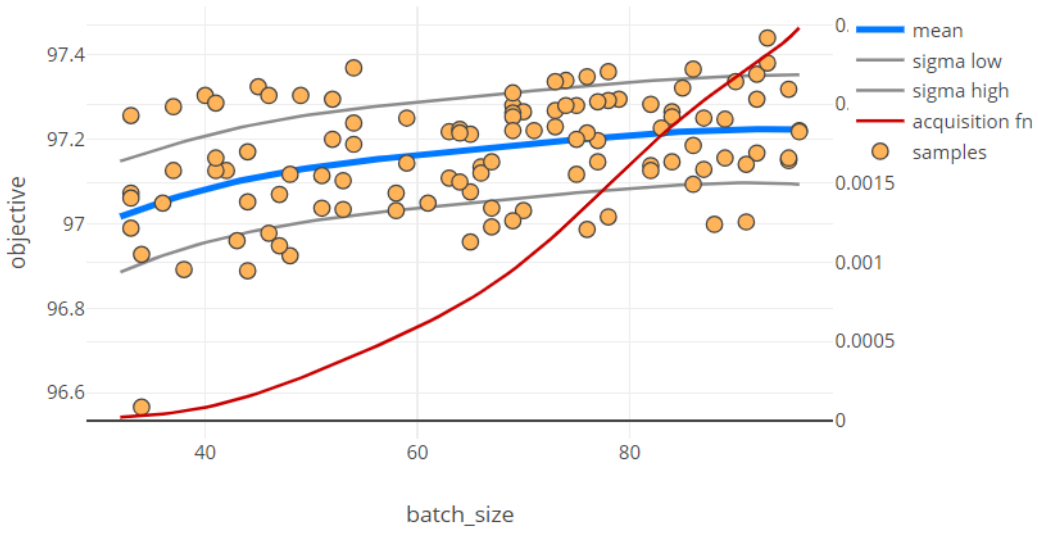


Figure 5.3: 1D marginal plot showing the effect of *batch size* on the objective function on the same model as shown in Figure 5.5. The red line shows the value of the acquisition function.

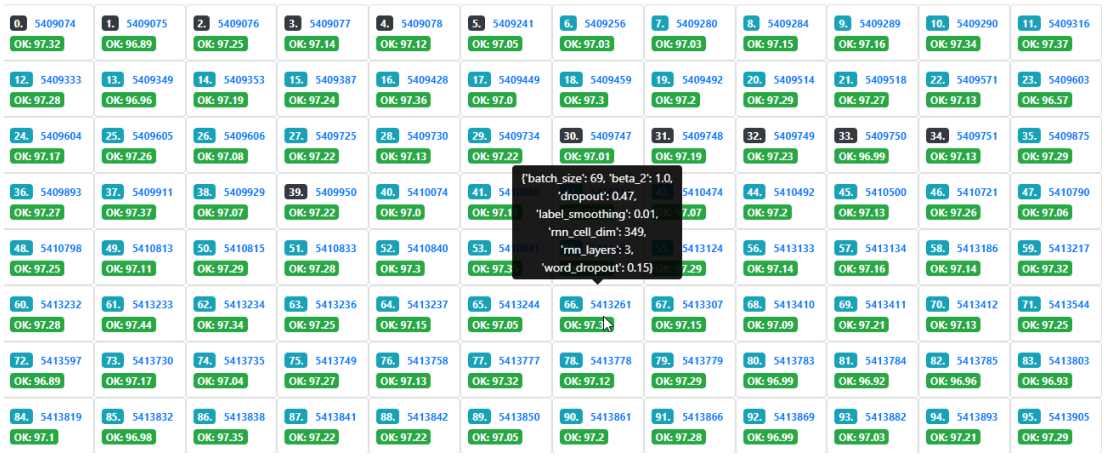


Figure 5.4: A timeline showing all of the evaluated experiments, together with their objective value, model type (shown in color), and the hyperparameters used for evaluation. The user can select any of the evaluations on the timeline and all of the plots will be shown from the perspective of that evaluation, i.e. what the model *saw* when choosing the hyperparameters of that specific evaluation.

saw at each point, and why it chose the hyperparameters it did.

### 5.5.1 Kernel Parameter Visualization

As a method of debugging possible issues in the model, as well as just general high level inspection of the quality of the fit, we provide a visualization of the kernel parameters as they change over time of the Bayesian optimization, as shown in Figure 5.5.

The kernel hyperparameters roughly determine the general properties of the regression curve and confidence intervals. A large lengthscale results in smooth functions, while small lengthscale results in many spikes without larger continuous regions. Plotting the value of each kernel parameter as the optimization progresses allows us to judge how much does the regression model's view of the objective function change over time.

Intuitively, after we have sampled enough data points, we would expect that adding one additional sample would have an effect on the regression curve itself, but not as much on its quality, such as smoothness. A large change in the kernel parameters signifies that it now sees the objective function completely differently than what it saw before. We would expect the parameters to change over time as the model refines itself to more data, but there should be a visible trend. In Figure 5.5 we show an example where for most of the samples the kernel parameters don't change by much, but there are two visible drops (around 45 and 55) in the *high_freq* lengthscale, which signify a possible issue in the GP regression at that point. Based on our experience with the model, we would still consider the example to be quite stable, as compared to another example shown in Figure 5.6, where until around 50 samples the model was not sure how to scale individual hyperparameters, and only stabilized afterwards. In this case we attribute the problem to a much noisier model.

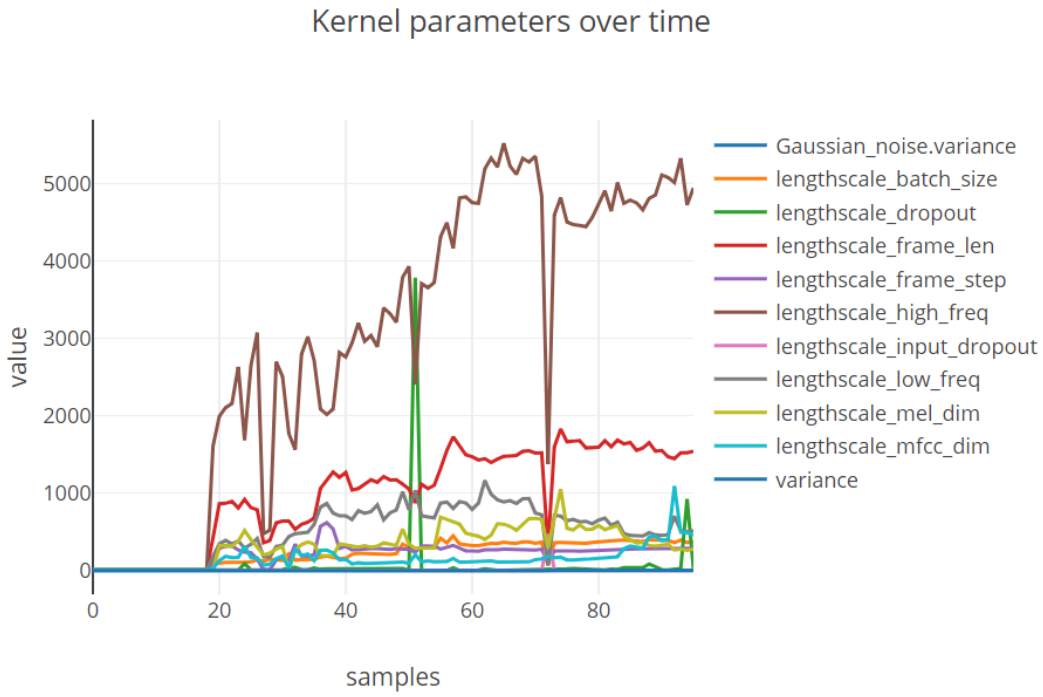


Figure 5.5: Visualization of the kernel parameters over time as the Bayesian optimization progresses. In this case the model was allowed one lengthscale $l$ parameter for each element of $\boldsymbol{x}$, in essence allowing it to normalize each column independently. Since there are 9 different hyperparameters the search space is 9-dimensional, and as a result it takes the model up to 20 samples to find a good fit in the data. After that, it automatically determines the scale of each parameter, such as the *high_freq* parameter, which was optimized on the scale of $1000 - 8000$. We can see the model clearly adapting its lengthscale to a similar range. As a general rule of thumb, when the lengthscale parameter is on the same order as the hyperparameter range, the model will scale that parameter to roughly unit range.

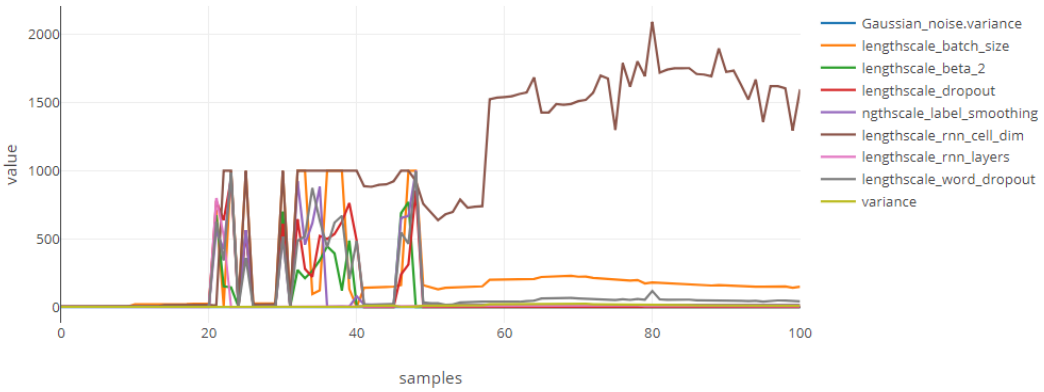


Figure 5.6: Visualization of a less stable model with kernel parameters rapidly changing until around 50 samples are evaluated. In this case we attribute the instability to a very noisy objective function, where the model took a long time to fit the right value of variance to counteract the effect of the noise.

### 5.5.2 Convergence

Looking at the results in a table does not always explain how well is the Bayesian optimization process doing. We add a convergence plot, which shows both the intermediate values of the objective, as well as a cumulative maximum.

In some cases the model might be achieving steady increase in the objective, as shown in Figure 5.5.2, while in others it might fail to improve in a significant way. The following Chapter 6 shows experiments where even though the model showed some improvement over the baseline, it clearly did not converge as more evaluations were computed, which becomes clearly visible in its convergence plot as shown in Figure 5.5.2.

## 5.6 Inspecting Attached Experiments

We provide results for some of the experiments as an attachment to this work. Because of the design described in Subsection 5.4.1 we only need to store the `meta.yml` file after the results have been collected, as the collection copies all of the results in.

Inspecting the results is then simply a matter of running either `bopt web -C dir` for the web interface, or `bopt exp -C dir` for the command line inspector, where `dir` is a directory containing the `meta.yml` file.

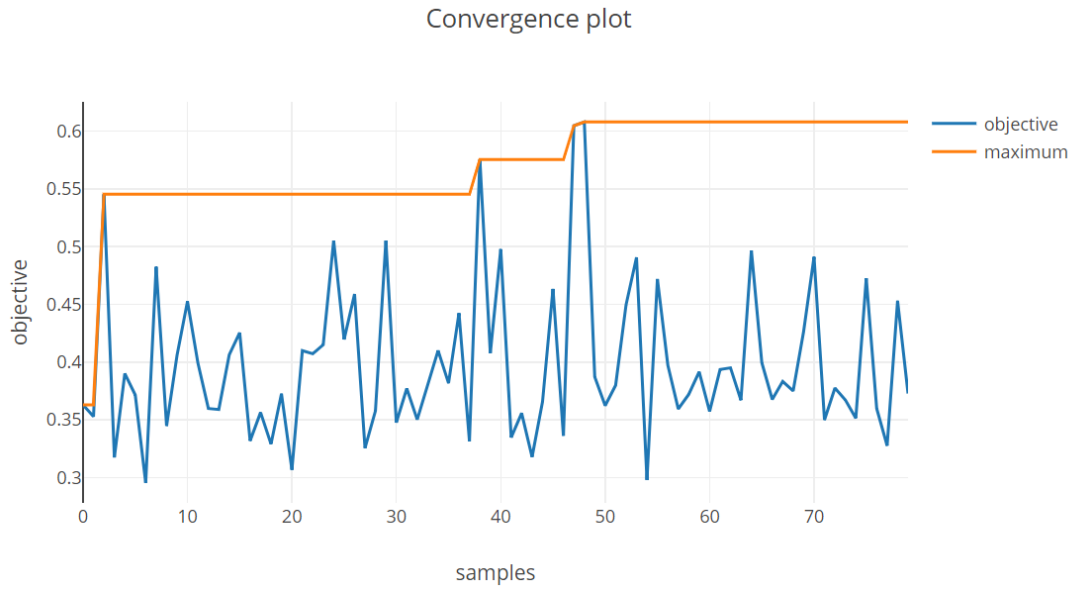


Figure 5.7: Visualization of the overall convergence of the optimization process. The $x$ axis labels time as new samples of are evaluated, and $y$ axis labels the objective function. We show both the intermediate results (blue), as well as the cumulative maximum (orange).

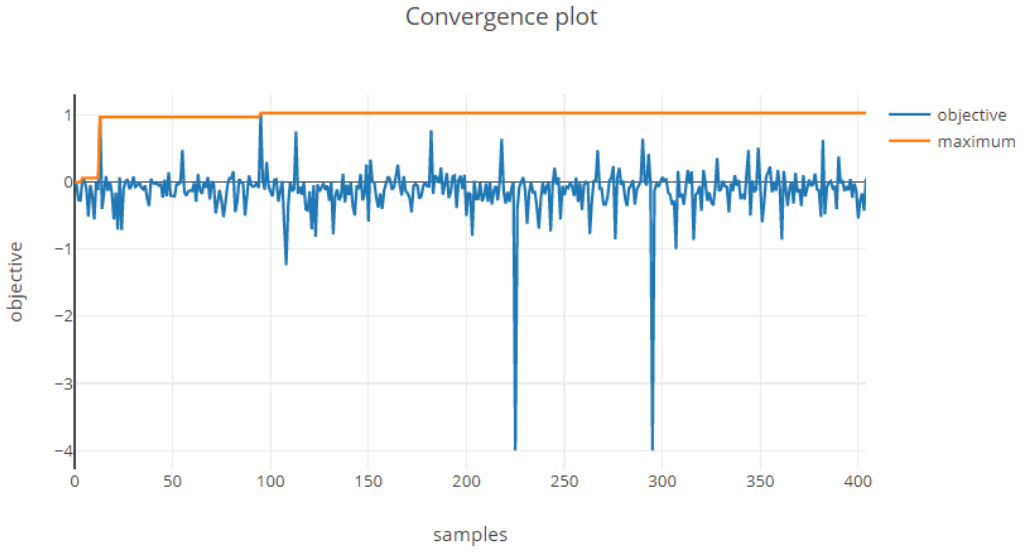


Figure 5.8: Convergence plot of a long running experiment where no significant improvement over the baseline is found. In this particular case the objective was measured as an improvement over an existing model.

# 6. Experiments

This chapter goes over some of the experiments we have conducted to evaluate the viability of Bayesian optimization. All of the cases are examples of optimizing neural networks, as that is the area of interest of this work.

We present one example from reinforcement learning, specifically the REINFORCE algorithm [Richard et al., 2018], as an example of a smaller and simpler network. This case also compares Bayesian optimization with random search. Next we show a larger experiment where we train a recurrent neural network (RNN, Goodfellow et al. [2016]), specifically a tagger, on a SIGMORPHON 2019 shared task [sig, a], where the network is trained on a significantly larger dataset. Lastly, we show two smaller examples, one of a tokenizer written in C++, and second a network for speech recognition.

## 6.1 REINFORCE

TODO: lower variance than random search?

### 6.1.1 Comparing Priors on Kernel Parameters

In this section we show the effect of setting kernel prior parameters from the data based on a simple heuristic.

TODO: empirical bayes

## 6.2 Tuning on Multiple Treebanks Simultaneously

In this larger experiment we tried to tune hyperparameters for a single model on multiple different datasets at once. Specifically, the model was trained on for the SIGMORPHON 2019 shared task 2 [sig, a] on 105 different treebanks, most of which are different languages. Our goal was to ideally find one set of hyperparameters that would work well across all of these treebanks. Even though this experiment was not ultimately successful we still include it in the text as the problem being solved poses interesting technical challenges from the perspective of hyperparameter tuning.

As a fundamental problem of this task, each treebank is of different size, and the difficulty of different languages varies a lot. To give a few specific examples, on Tamil the model achieves around 95% accuracy in tagging, while on Sanskrit it achieves only 65%. This makes it nearly impossible to set the objective function to simply be the accuracy of the model, as the same value of hyperparameters could result in the objective varying by 30%.

We work around this problem by computing a baseline accuracy using an existing, hand tuned, model on the same task, pre-computing its score for each treebank with a fixed set of hyperparameters, and then subtracting its value from the accuracy of our tuned model. The objective function then explicitly becomes the improvement over an existing fixed model.

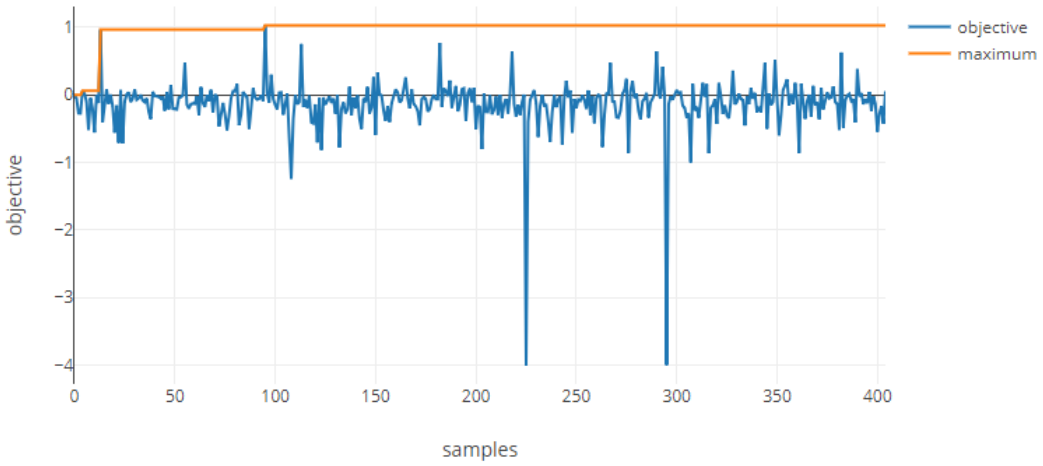


Figure 6.1: Convergence plot for a model trained on the SIGMORPHON 2019 shared task. No significant improvement over the baseline was found.

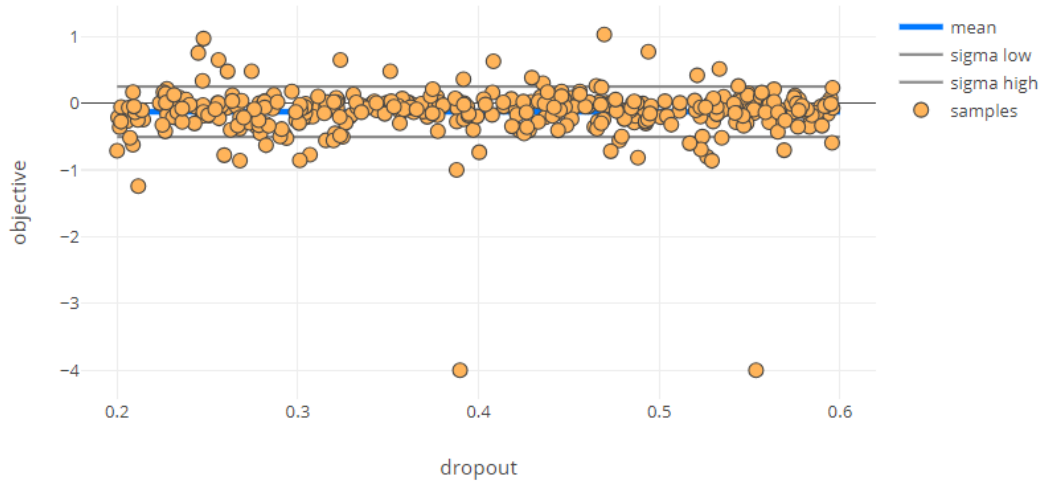


Figure 6.2: Marginal regression on the dropout hyperparameter for the SIGMORPHON 2019 shared task when training on all 105 treebanks.

A second problem we had to solve was how to incorporate the 105 treebanks into the optimization procedure. Because training the model takes multiple GPU hours even for the smaller treebanks we realistically cannot perform more than a few hundred evaluations total. Instead of training the model on all treebanks for each set of hyperparameters we only train the model on one treebank for each hyperparameter configuration chosen by Bayesian optimization.

This introduces an interesting choice. In theory, we could treat the treebank as a hyperparameter treated explicitly by Bayesian optimization. As the algorithm balances the exploration-exploitation tradeoff it should be able to pick treebanks as needed to better explore the space. But because of the different sizes and difficulties of each treebank, and the inability of Bayesian optimization to treat categorical hyperparameters well, we decided to not take this approach.

Instead, we omit the treebank from the list of hyperparameters so that the optimizer has no way of modeling it, and provide it as an explicit parameter outside of the scope of the optimizer. In theory, we could also provide the optimizer with a treebank that was used with each valuation and not allow it to optimize it when choosing a next point by overriding its choice, but this approach would most likely yield sub-optimal choices by the optimizer, as the samples would be chosen based on a different criteria than what the optimizer sees. Instead, our way of completely removing the treebank parameter from the optimization process will be seen by the optimizer as noise on the output. If it were to evaluate the same hyperparameters multiple times, the evaluation itself would receive a different treebank, and the improvement over the baseline would be different. In effect this captures our desire to find a set of hyperparameters that works well across all treebanks.

Unfortunately, our experimental results did not find a significant improvement over the hyperparameters found by hand-tuning the network (see Figure 6.1). One of the reasons is that the size of the treebank affects which hyperparameter combinations result in value of the objective function. An example of using tuning the same model on a larger treebank is shown in Section 6.3.

To give a specific example, when the *batch_size* hyperparameter is set to a larger value, it will work well on a larger treebank, but the opposite is true for a smaller treebank, where a smaller *batch_size* acts as a regularizer. This means our samples at the same batch size will be spread as far as is the objective on all treebanks. Unfortunately, this spread ends up being so large that all of the hyperparameter end up having no visible trend, as no value is clearly better than the others, as shown in Figure 6.2.

## 6.3 Tagger on a Single Treebank

Contrary to the experiment shown in Section 6.2 in this experiment we train the same model but only on a single treebank at a time. We tried two different treebanks, Czech and English, which are both larger. We chose the larger treebanks because they provide a more consistent evaluation metric, as the smaller ones are many orders of magnitude smaller, and pose a greater risk of overfitting even on the validation set.

While the previous experiment (see Section 6.2) did not find any trends in the hyperparameters, the one we present here did find a clear trend in most, as

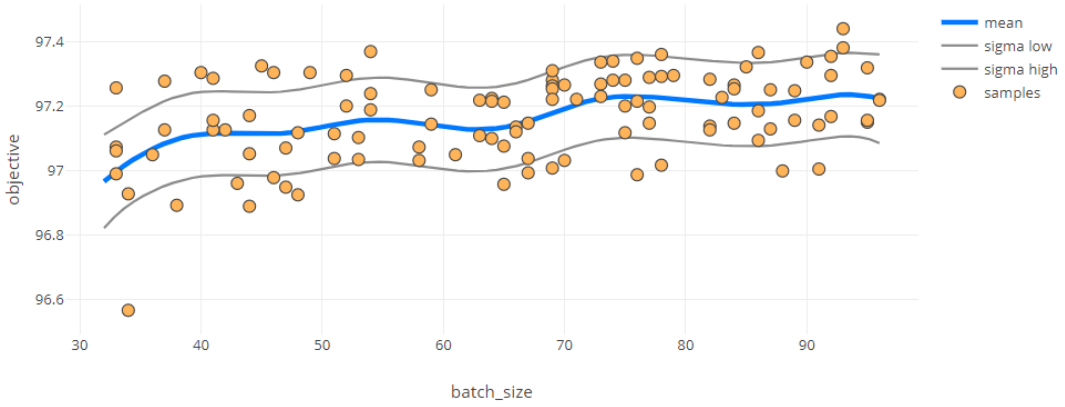


Figure 6.3: Marginal regression on the *batch_size* hyperparameter for the SIGMORPHON 2019 shared task when training on a single Czech treebank.

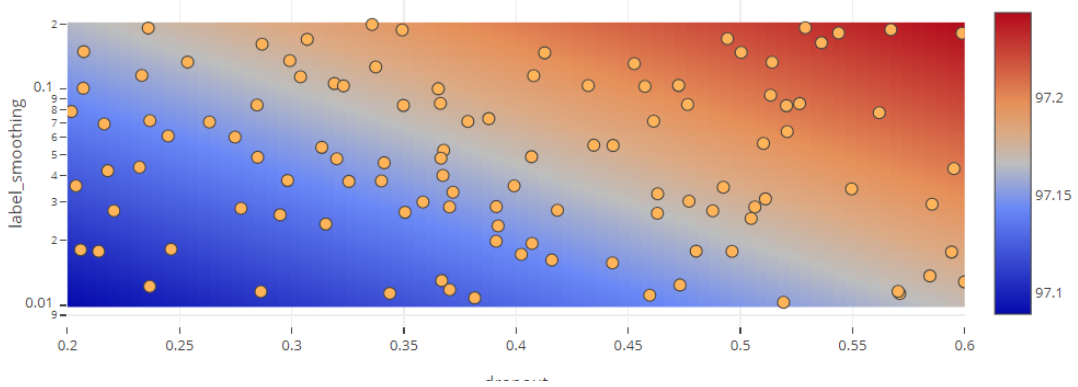


Figure 6.4: Marginal regression on the *label_smoothing* and *dropout* hyperparameters for the SIGMORPHON 2019 shared task when training on a single Czech treebank. A correlation between the two parameters is clearly visible, showing that setting only one of them to the correct value is not enough to achieve a high value of the objective function.

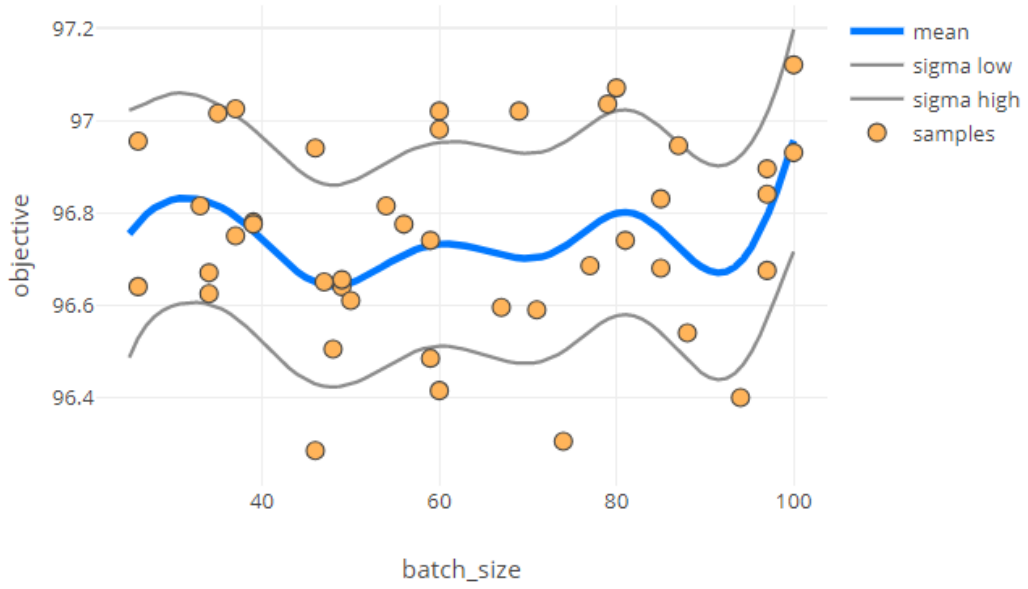


Figure 6.5: GP regression overfitting on the marginal distribution of *batch_size*.

shown for example in Figure 6.3, where larger *batch_size* is better. In some cases we also found clear relationships between two hyperparameters, such as in the case of *label_smoothing* and *dropout* shown in Figure 6.4

## 6.4 Other Smaller Experiments

In this section we briefly show some interesting results from our last two experiments. First, let us show the experiment where we optimized hyperparameters of UDPipe [Straka and Straková, 2017] on tokenization of Czech, English and Ancient Greek.

Some of the interesting result in this case are the failure cases of the GP regression, particularly how it can overfit on seemingly uninformative dataset, as shown in Figure 6.5. These examples nicely show the high capacity of a GP which, even when we set informative priors, is able to find a surprising way to fit the data with high likelihood. Despite these problems, the Bayesian optimization approach still managed to find hyperparameter values with an objective very close to that find by an exhaustive grid search. Because of the acquisition function, even if the model overfits and picks a bad sample, it will be able to correct itself right afterwards, as the area that was previously receiving high value of acquisition function was explored.

As a result, the model still might end up wasting computation time by evaluating incorrect areas by overfitting on the data, but we have not observed it to get stuck in a local optima. Because Bayesian optimization explicitly models exploration into its decision making it quite often ends up doing a decent job of exploring most of the search space, as is visible in all of the figures.

- maly ulohy
- velka uloha

- interpretace vysledku

# 7. Conclusion

- future work
    - pribalit vysledky experimentu a jak jednoduse pustit web

- parser - tokenizer/segmentace - speech recognition - opennmt lemmatizace - reinforce_with_baseline

# Conclusion

# Bibliography


Amazon sagemaker. https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning-how-it-works.html. Accessed: 2019-04-30.

Automl. https://www.automl.org/automl/. Accessed: 2019-04-30.

OpenAI five. https://openai.com/five/. Accessed: 2019-04-22.

paramz. https://github.com/sods/paramz. Accessed: 2019-04-27.

SGE. https://arc.liv.ac.uk/trac/SGE. Accessed: 2019-04-27.

Sigmorphon 2019 task 2: Morphological analysis and lemmatization in context. https://sigmorphon.github.io/sharedtasks/2019/task2/, a. ACcessed: 2019-05-02.

Sigopt. https://sigopt.com/product/optimization-engine/, b. Accessed: 2019-04-30.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt, 2016.

C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 2016. ISBN 9781493938438. URL https://books.google.cz/books?id=kOXDtAEACAAJ.

David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

George EP Box. A note on the generation of random normal deviates. *Ann. Math. Stat.*, 29:610–611, 1958.

Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010. URL http://arxiv.org/abs/1012.2599.

Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. Dealing with integer-valued variables in bayesian optimization with gaussian processes. *arXiv preprint arXiv:1706.03673*, 2017.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017. URL `http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization`.

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

GPy. GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`, since 2012.

Harvard Intelligent Probabilistic Systems Group. Spearmint. `https://github.com/HIPS/Spearmint`, 2014.

Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. scikit-optimize/scikit-optimize: v0.5.2, March 2018. URL `https://doi.org/10.5281/zenodo.1207017`.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`. [Online; accessed <today>].

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018. URL `http://arxiv.org/abs/1812.04948`.

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

A. Klein, S. Falkner, N. Mansur, and F. Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, December 2017.

Aravindh Krishnamoorthy and Deepak Menon. Matrix inversion using cholesky decomposition. In *2013 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 70–72. IEEE, 2013.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997. ISBN 0070428077. URL `https://www.amazon.com/Machine-Learning-Tom-M-Mitchell/dp/0070428077?SubscriptionId=`

AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0070428077.

K.P. Murphy and F. Bach. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machi. MIT Press, 2012. ISBN 9780262018029. URL https://books.google.cz/books?id=NZP6AQAAQBAJ.

Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pages 123–137. Springer International Publishing, 2016. ISBN 978-3-319-31204-0. doi: 10.1007/978-3-319-31204-0_9. URL http://dx.doi.org/10.1007/978-3-319-31204-0_9.

Sutton Richard, BARTO SUTTON, and G ANDREW. *Reinforcement learning: an Introduction*. MIT Press, 2018.

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

Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

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

Milan Straka and Jana Straková. Tokenizing, pos tagging, lemmatizing and parsing ud 2.0 with udpipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada, August 2017. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/K/K17/K17-3009.pdf.

S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2): 22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37.

G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT Press Cambridge, MA, 2006.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017. URL http://arxiv.org/abs/1707.07012.

# List of Figures