IMPROVING BAYESIAN OPTIMIZATION FOR MACHINE LEARNING
USING EXPERT PRIORS

by

Kevin Swersky

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2017 by Kevin Swersky
Abstract

Improving Bayesian Optimization for Machine Learning using Expert Priors

Kevin Swersky
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2017

Deep neural networks have recently become astonishingly successful at many machine learning problems such as object recognition and speech recognition, and they are now also being used in many new and creative ways. However, their performance critically relies on the proper setting of numerous hyperparameters. Manual tuning by an expert researcher has been a traditionally effective approach, however it is becoming increasingly infeasible as models become more complex and machine learning systems become further embedded within larger automated systems. Bayesian optimization has recently been proposed as a strategy for intelligently optimizing the hyperparameters of deep neural networks and other machine learning systems; it has been shown in many cases to outperform experts, and provides a promising way to reduce both the computational and human time required. Regardless, expert researchers can still be quite effective at hyperparameter tuning due to their ability to incorporate contextual knowledge and intuition into their search, while traditional Bayesian optimization treats each problem as a black box and therefore cannot take advantage of this knowledge. In this thesis, we draw inspiration from these abilities and incorporate them into the Bayesian optimization framework as additional prior information. These extensions include the ability to transfer knowledge between problems, the ability to transform the problem domain into one that is easier to optimize, and the ability to terminate experiments when they are no longer deemed to be promising, without requiring their training to converge. We demonstrate in experiments across a range of machine learning models that these extensions
significantly reduce the cost and increase the robustness of Bayesian optimization for automatic hyperparameter tuning.
Acknowledgements

This thesis, and all of my work beyond it, are the product of the incredible guidance and support that I have received, and the lasting friendships that I have made throughout my time in academia. I cannot stress enough the many ways in which the people in my life have inspired me. To these people I express my deepest gratitude.

First I absolutely must acknowledge my supervisor Richard Zemel. Rich would never hesitate to entertain a new idea, no matter how silly. He would happily dive head first into the depths of a new, unknown area, become an expert, and help me through with excellent advise. Beyond this, Rich was an incredible role model. He carries himself with a great deal of humility, and is respectful to everyone around him. I was unbelievably lucky to have had the opportunity to work with him, and I hope that in my career I can be even a fraction as good a mentor as he is.

I would like to thank my supervisory committee, Geoffrey Hinton and Brendan Frey, for their excellent feedback and support throughout my degree. I had the privilege to work with Brendan on multiple projects and was always energized by his constant enthusiasm and keen ideas. I was incredibly fortunate to be able to work with Geoff. His intuition and ability to get to the core insight of any problem is astounding, but even more important is his generosity and the way he treats those around him. I was also fortunate to work with or alongside many excellent professors, including Sanja Fidler, Toniann Pitassi, Ruslan Salakhutdinov, and Raquel Urtasun. I would also like to thank my external examiner Max Welling for his thoughtful and insightful comments on this thesis.

I was fortunate enough to have several other excellent mentors during my degree. Early on I worked a lot with Danny Tarlow, who helped me build research momentum and reach a new level of maturity in my work. Later, I collaborated with Jasper Snoek, whose unwavering work ethic, novel insights, and homemade beer led to the papers that form the foundation of this thesis. Throughout all of this, I was privileged to be able to collaborate with Ryan Adams, who oversaw much of my work on Bayesian optimization. Ryan continues to astonish me with his breadth and depth of knowledge, and by his kindness and sensitivity. I am indebted to my former advisor Nando de Freitas, who brought me into the world of machine learning, advised me during my master’s degree, brought me back into the world of machine learning after I graduated, and continues to be a good friend to this day.

Perhaps the greatest aspect of my time in academia is the group of friends that I worked with, socialized with, and crunched with at deadlines. These people are unbelievably talented, smart, genuine, and drove me to set the bar as high as I possibly could.
They include Jimmy Ba, Yuri Burda, Laurent Charlin, George Dahl, David Duvenaud, Roman Garnett, Michael Gelbart, Alex Graves, Roger Grosse, Frank Hutter, Navdeep Jaitly, Emtiyaz Khan, Jamie Kiros, Alex Krizhevsky, Hugo Larochelle, Yujia Li, Chris Maddison, Ben Marlin, James Martens, Vlad Mnih, Abdel-rahman Mohamed, Michael Osborne, Mengye Ren, Oren Rippel, Mark Schmidt, Alex Schwing, Jake Snell, Nitish Srivastava, Ilya Sutskever, Charlie Tang, Tijmen Tieleman, Maks Volkovs, Jackson Wang, Alex Wiltschko, and Youssef Zaky. This is certainly not an exhaustive list.

Finally, I owe a debt of gratitude to those closest to me, whose endless love saw me through this degree and everything leading up to it. I want to thank my girlfriend Sarah Austin, for always being there for me through even the most trying times, and for all of the adventures we shared along the way. I am grateful for the support of my stepdad Michael, my older brother Darren, my younger brother Lorne, and my stepsister Ashley. I would like to thank my stepbrother Nathan for pushing me to move outside of my comfort zone and for always lending an ear when I needed one. I am grateful for the short time I had with my late father Gerald, for the memories and values he bestowed upon me during our time together. Finally, I am most thankful to my dear mother Joanne, who took on an incredible burden of being a single, working mother and made it look easy. This thesis is as much her hard work as it is mine.
# Contents

1 Introduction 1  
1.1 Summary of Thesis Chapters 3  

2 Background 5  
2.1 Deep Learning for Classification 5  
2.1.1 Logistic Regression 5  
2.1.2 Multi-layer Perceptrons 6  
2.1.3 Convolutional Neural Networks 7  
2.2 Training Deep Models 8  
2.2.1 Stochastic Gradient Descent 9  
2.2.2 Regularization 10  
2.3 Collaborative Filtering 12  
2.3.1 Probabilistic Matrix Factorization 12  
2.4 Topic Modelling 13  
2.4.1 Online Latent Dirichlet Allocation 14  
2.5 Types of Hyperparameters 14  
2.6 Bayesian Optimization 15  
2.6.1 Gaussian Processes 16  
2.6.2 Acquisition Functions 20  
2.6.2.1 Pure Exploration and Exploitation 21  
2.6.2.2 Probability of Improvement 21  
2.6.2.3 Expected Improvement 22  
2.6.2.4 Entropy Search 22  
2.6.3 Practical Considerations 24  
2.6.3.1 Learning Gaussian Process Hyperparameters 24  
2.6.3.2 Initializing the Search 24  
2.6.3.3 Optimizing the Acquisition Function 24  
2.6.3.4 Integrated Acquisition Functions 25
2.6.4 Extensions .................................................. 25
   2.6.4.1 Constraints .......................................... 25
   2.6.4.2 Parallelism ......................................... 27

3 Multi-Task ..................................................... 28
   3.1 Introduction ............................................. 28
   3.2 Multi-Task Gaussian Processes ......................... 29
   3.3 Transferring Knowledge to a New Task .................. 30
   3.4 Optimizing an Average Function over Multiple Tasks .. 31
   3.5 A Cost-Sensitive Multi-Task Acquisition Function .... 33
   3.6 Experiments .............................................. 37
      3.6.1 The Cold Start Problem ......................... 37
      3.6.2 Fast Cross-Validation ......................... 38
      3.6.3 Using Small Datasets to Bootstrap Training on Large Datasets .. 39
   3.7 Discussion and Conclusion .............................. 40

4 Input Warping ................................................. 45
   4.1 Introduction ............................................. 45
   4.2 Non-stationary Gaussian Process Regression ............ 46
      4.2.1 Input Warping .................................. 46
      4.2.2 Integrating over Warpings ..................... 47
      4.2.3 Multi-Task Input Warping ....................... 50
      4.2.4 Other Methods for Non-Stationary Gaussian Process Regression .. 50
   4.3 Experiments .............................................. 51
      4.3.1 Comparison to Stationary GPs .................. 51
      4.3.2 HPOLib Continuous Benchmarks .................. 53
      4.3.3 Multi-Task Warping Experiments ............... 55
   4.4 Discussion .............................................. 57
      4.4.1 The Boundary Issue .............................. 57
      4.4.2 Future Directions .............................. 58
   4.5 Conclusion .............................................. 58

5 Architecture Search ......................................... 60
   5.1 Introduction ............................................. 60
   5.2 Binarizing Conditional Variables ....................... 61
   5.3 Simple Solutions ....................................... 61
   5.4 A Kernel for Conditional Parameter Spaces ............ 62
5.4.1 Intuitive Properties of a Conditional Embedding .......................... 63
5.4.2 Cylindrical Embedding ............................................................. 63
5.5 Experiments ................................................................................. 65
5.6 Discussion .................................................................................. 68
5.7 Conclusion and Future Directions ............................................... 68

6 Freeze-Thaw .................................................................................. 70
  6.1 Introduction ................................................................................ 70
    6.1.1 A Kernel for Training Curves ............................................... 71
    6.1.2 An Efficient Hierarchical Gaussian Process ......................... 74
      6.1.2.1 Specification ............................................................... 74
      6.1.2.2 Inference ................................................................. 75
      6.1.2.3 Computational Complexity ........................................... 78
    6.1.3 The Freeze-Thaw Strategy .................................................... 78
  6.2 Experiments ................................................................................ 80
    6.2.1 Models ................................................................................. 81
    6.2.2 Results ................................................................................. 82
  6.3 Conclusion .................................................................................. 82

7 Conclusion ...................................................................................... 86

8 Appendix .......................................................................................... 90
  A Multivariate Gaussian Distribution Identities .............................. 90
    A.1 Density .................................................................................. 90
    A.2 Marginalization and Conditioning ......................................... 90
    A.3 Affine Transformations .......................................................... 91
    A.4 Matrix Inversion and Determinant Lemmas ............................ 91
  B Multi-Task Experiment Details .................................................... 92
    B.1 Cold Start Experiments .......................................................... 92
  C Freeze-Thaw Implementation Details ........................................ 93
    C.1 Kernel and GP Hyperparameters ........................................... 94

Bibliography ....................................................................................... 95
## List of Tables

4.1  A comparison of hyperparameter optimization methods on the continuous benchmark problems from [Eggensperger et al., 2013]. ........................................... 55

4.2  Number of evaluations required for warping to converge to the optimal solution. ................................................................. 55
List of Figures

2.1 Functions drawn from a (a): squared exponential, (b) Matérn-$\frac{3}{2}$, (c): Matérn-$\frac{5}{2}$ plus noise and (d): Ornstein-Uhlenbeck kernel. Each color is drawn using the same seed in order to highlight the differences between the kernels. Different colors correspond to different length scales. We use $\theta_1 = 1$ for all kernels. Best viewed in colour. ............................................ 19

3.1 An example of how multi-task GPs can be used to improve Bayesian optimization. .......................................................... 31

3.2 An example of entropy search with (a): uncorrelated tasks, and (b): correlated tasks. Figure (c) shows the cost-sensitive multi-task entropy search acquisition function. .......................................................... 36

3.3 (a)-(b): validation error per function evaluation. (e), (f): average cumulative error (ACE) over function evaluations. The dashed black line is the best settings from the initial task applied to the new task. ........................................ 42

3.4 (a): PMF cross-validation error per function evaluation on Movielens-100k. (b): lowest error observed for each fold per function evaluation for a single run. .......................................................... 43

3.5 (a), (c): validation error as a function of time spent training the models. (b), (d): validation error over the number of function evaluations. (e), (f): time taken to reach a given validation error. ........................................ 44

4.1 Two examples of how input warping using the beta CDF can transform a non-stationary function into a stationary one. The warping function maps the original inputs on the horizontal axis to new inputs shown on the vertical axis. The effect is to stretch and contract regions of the input space in such a manner as to remove non-stationarity. ........................................ 47
4.2 Each figure shows 50 warping functions resulting from the beta CDF where the shape parameters $\alpha$ and $\beta$ are sampled from a log-normal prior with a different mean and variance. The flexible beta CDF captures many desirable warping functions and adjusting the prior over input warping allows one to easily encode prior beliefs over the form of non-stationarity. The parameters of the prior distribution, expressed as $(\mu_\alpha, \sigma^2_\alpha, \mu_\beta, \sigma^2_\beta)$ are, (a): (0.0, 0.75, 0.0, 0.75), (b): (1.5, 0.75, 0.0, 0.75), (c): (0.0, 0.75, 1.5, 0.75),
(d): (1.5, 0.75, 1.5, 0.75), (e): (−1.0, 0.75, −1.0, 0.75).

4.3 An empirical comparison of Bayesian optimization following the standard Gaussian process expected improvement algorithm (GP EI MCMC) and our strategy (Warped GP EI MCMC) for modeling input non-stationarity. The methods are compared on four challenging problems involving the optimization of the hyperparameters of popular machine learning algorithms.

4.4 Example input warpings learned for the logistic regression problem (Figures (a), (b)) and the parameters of the deep convolutional neural network (Figures (c), (e), (d)). Each plot shows the mean warping, averaged over 100 samples, of each of the parameters. Figure (b) shows the warping learned on the learning rate parameter for logistic regression with different numbers of observations, along with the standard deviation. Each curve in Figures (d) and (e) is annotated with the depth of the layer that each parameter is applied to.

4.5 Multi-task warping applied to logistic regression and online LDA. For logistic regression, the model is trained on USPS first and then the search is transferred to MNIST. For online LDA, the data is subsampled and the model is learned, then the search transferred to the full dataset. In Figure 4.5a, we show the mean warping learned by each task for each parameter. The solid lines indicates the MNIST task, while the dashed lines indicate the USPS task.

5.1 An example of converting an integer-valued conditional variable (depth) in an MLP into a sequence of boolean conditional variables.
5.2 A demonstration of the embedding giving rise to the pseudo-metric. All points for which \( \delta_2(x) = \text{false} \) are mapped onto a line varying only along \( x_1 \). Points for which \( \delta_2(x) = \text{true} \) are mapped to the surface of a semicylinder, depending on both \( x_1 \) and \( x_2 \). This embedding gives a constant distance between pairs of points which have differing values of \( \delta \) but the same values of \( x_1 \).

6.1 Example functions drawn from the exponential decay kernel, where the x-axis represents time. (a) Example basis functions with \( \alpha = 1.0 \) and \( \beta = 0.5 \). These basis functions take the form \( \exp(-\lambda t) \). The curves are obtained by taking uniformly spaced values between 0 and 1 and then passing them through the inverse CDF of the Gamma distribution, parameterized by \( \alpha \) and \( \beta \), to get the values of \( \lambda \). (b) Samples from a Gaussian process with this covariance function. (c) Samples from a Gaussian process conditioned on the curves starting at a positive number and with an added Ornstein-Uhlenbeck kernel to simulate natural training curves.

6.2 (a): Factor graph representation of the GP model for training procedures. Each row represents a learning curve that is drawn from an independent GP prior, conditioned on its mean. The mean of each learning curve is jointly drawn with the mean of the other curves using another GP prior. (b): Partially completed training curves, and the GP prediction at their eventual asymptotes. (c): The posterior GP prediction at the asymptote. Each colored point represents the GP prediction at the hyperparameter location corresponding to a training curve with the same color.

6.3 This figure shows the results of the empirical comparison to standard Bayesian optimization on four common machine learning hyperparameter optimization problems. For each problem we report the lowest loss observed over all training epochs evaluated by each method, averaged over five optimization runs.
6.4 A visualisation of the progression of the optimization curves throughout the optimization procedure on the probabilistic matrix factorization problem. Figure 6.4a shows for each distinct hyperparameter setting evaluated, the optimization curve run out by the procedure. The curves are ordered by the iteration in the Bayesian optimization procedure that they were started and each epoch of each curve is colored by the iteration of the Bayesian optimization that this section was evaluated. From the figure we can see that the procedure frequently stops running training curves that are not promising and often returns to promising training curves that were previously started. Figure 6.4b shows a two dimensional cross section of Figure 6.4a.
Chapter 1

Introduction

Machine learning is becoming an increasingly relevant topic in computer science. It is now used to solve many difficult problems, from computer vision [Krizhevsky et al., 2012], to speech recognition [Hinton et al., 2012a], to movie recommendations [Netflix, 2009], to machine translation [Sutskever et al., 2014]. Moreover, machine learning is now impacting many areas beyond these, such as medicine [Shin et al., 2013], genetics [Alipanahi et al., 2015], drug discovery [Dahl et al., 2014], even art [Gatys et al., 2015] and games [Mnih et al., 2015]. Much of this revolution is being driven by a particular form of machine learning called deep learning in which a neural network learns multiple layers of nonlinear features from data that allow them to generalize across many domains and problems.

Virtually every machine learning model comes with an associated set of hyperparameters. Hyperparameters govern the high-level properties of a model. The term hyperparameter is borrowed from the field of nonparametric Bayes, where they are used to specify priors on hierarchical Bayesian models. In general machine learning, hyperparameters tend to impose a prior on the capacity and learning behaviour of the model.

Hyperparameters are different from model parameters in that they generally do not admit efficient learning algorithms and are usually set by a form of trial-and-error called cross-validation; a proxy for generalization performance. Each iteration of hyperparameter tuning requires the model to be trained so that the validation performance can be measured. In the modern world of massive datasets and computationally intensive neural networks, this is a painstakingly expensive procedure.

Traditional methods for hyperparameter tuning involve using intuition and heuristics. Manual tuning remains a common technique, but it requires an expert to invest a great deal of time building an intuition for how the hyperparameters affect the model—time that would be better spent exploring new models. Other heuristics include grid search
and random search, which use very little information about the hyperparameter-model relationship and can waste a great deal of computation exploring inferior settings.

As a remedy for this issue, researchers have turned to the idea that hyperparameter tuning can be treated explicitly as a global optimization problem. A popular family of research in this direction uses the idea of response surface methods [Box and Wilson, 1951]. These methods build a fast, surrogate model to approximate the hyperparameter to validation performance relationship and uses this to guide the search toward promising settings.

An appealing form of response surface methods is known as Bayesian optimization [Jones et al., 1998]. It is appealing because it uses a principled characterization of uncertainty based on Bayes rule to govern the exploration-exploitation tradeoff, the balance between trying settings with which we have very little information, and settings that we are confident will yield good results. The Bayesian optimization approach has led to a number of successful outcomes in tuning machine learning and deep learning models [Snoek et al., 2012].

Despite these successes, traditional Bayesian optimization suffers from a number of shortcomings when compared to expert intuition. In particular, experts develop a deep understanding of their models and how they are affected by hyperparameter choices. This allows them to exploit very strong prior information when using manual tuning. For example, when tuning the same model on multiple datasets, an expert will usually assume that settings that worked on one dataset will perform reasonably well on another. They can also probe the learning process by inspecting learning curves to detect divergence or overfitting, which allows them to reason about which settings to try next. Traditional Bayesian optimization, on the other hand, must learn about the problem entirely through trial-and-error.

The goal of this thesis is to build these sorts of priors directly into the Bayesian optimization framework in order to improve the optimization efficiency. To do this, we investigate ways in which Gaussian processes—the model on which Bayesian optimization is typically based—can be modified in order to take advantage of additional structure in the optimization problem. We also investigate how to modify the utility functions from which the exploration-exploitation tradeoff is made.

In this thesis, we show how the Bayesian optimization framework can be significantly extended to incorporate priors based on expert intuition, while still maintaining the principled characterization of uncertainty. Across a range of machine learning models, we show how these extensions improve both the speed and robustness of Bayesian optimization.
1.1 Summary of Thesis Chapters

- In Chapter 2, we provide the foundational knowledge for modern hyperparameter tuning. We introduce the models that we will frequently refer to and experiment with in this thesis, including deep neural networks and convolutional neural networks, along with other popular algorithms such as probabilistic matrix factorization and latent Dirichlet allocation. We introduce the Bayesian optimization algorithm, from the underlying Gaussian process model to different acquisition functions for choosing which hyperparameters to test. We discuss extensions of Bayesian optimization, such as parallelism and constraints, that help make Bayesian optimization more practical in a real-world setting.

- In Chapter 3, inspired by the ability of experts to transfer knowledge between problems, we describe the multi-task Bayesian optimization algorithm. This is an extension of Bayesian optimization that allows similar optimization problems to share information with each other in order to speed up each individual optimization. A common example where this arises is optimizing the hyperparameters of a single model on multiple datasets. We show how this approach can be used to solve the cold-start problem, where an optimization can be initialized with information from other problems. We use this technique to introduce a fast cross-validation method, where cross-validation can be performed while only evaluating a single fold per hyperparameter setting. We end the chapter by introducing a method to use a cheap, auxiliary task in order to provide additional information that can be used to rapidly optimize an expensive, primary task. This work appears in [Swersky et al., 2013b].

- Chapter 4 describes input warping for Bayesian optimization. Experts will often transform hyperparameters before tuning in order to remove non-stationary effects. Non-stationarity is when the function changes at different rates throughout the hyperparameter space. Traditional Bayesian optimization assumes stationarity, and this can hurt the optimization efficiency. Our solution is to learn a simple family of parametric warping functions that capture common non-stationary behaviours. We show that this can yield substantial improvements to the speed and robustness of Bayesian optimization across many different problems. We further show how this can be used to improve multi-task Bayesian optimization by learning a separate warping for each task. This work appears in [Snoek et al., 2014].

- We address the problem of architecture search in Chapter 5. An architecture is a
particular instantiation of a model that carries with it a particular set of hyperparameters. Different architectures are different instantiations of the same model that carry distinct, but potentially overlapping sets of hyperparameters. An example of this is a neural network model with a varying number of layers. For each setting of the number of layers, the neural network can have a different set of hyperparameters belonging to each distinct layer. In an ideal setting, we would be able to learn the similarity between different architectures and use this to find both the best architecture and the best hyperparameter settings, much like in a multi-task setup. Traditional Gaussian process models do not typically deal with this problem as it involves comparing variable-length inputs. In this chapter, we show how a kernel based on embeddings of these vectors to a fixed-sized space can be used to alleviate this issue. This work appears in [Swersky et al., 2013a].

• Experts will often investigate the performance of a model as it is training, not just at the end of training, by inspecting the learning curves. In this way, they can quickly eliminate poorly performing models and devote the computational cycles to new, more promising models. In Chapter 6, we introduce a Bayesian optimization system that is capable of mimicking this behavior. We derive a new form of Gaussian process model that is designed specifically to predict optimization curves, while scaling gracefully in terms of computational expense. We use this within an information theoretic decision framework in order to determine which models to try, and which models to terminate. We show how this ability to perform early stopping of poor hyperparameter settings can yield substantial improvements to the optimization speed. This work appears in [Swersky et al., 2014].

• In Chapter 7 we conclude by summarizing the contributions of this thesis, discussing broader perspectives, and outlining potential future directions of research.
Chapter 2

Background

2.1 Deep Learning for Classification

In this section we will outline some of the basic deep models that we experiment with for classification problems. Classification, particularly visual object recognition, is an extremely popular domain for deep learning as deep models tend to dramatically outperform shallow models trained on carefully constructed pre-defined features [Krizhevsky et al., 2012]. The caveat is that they are computationally expensive; deep models tend to work best with very large datasets and require many layers of dense matrix multiplication and spatial convolutions.

In multi-label classification, we assume that we are given input and observation pairs of the form \((x, y)\), where \(y \in \{e_c\}_{c=1}^C\) where \(e_c\) is a \(C\)-dimensional elementary vector where element \(c\) is 1 and the rest of the elements are 0. The domain of \(x\) is flexible, usually consisting of real variables, integers, or categorical variables (represented as elementary vectors).

2.1.1 Logistic Regression

Logistic regression is a linear classification method. It is shallow rather than deep, in the sense that it operates directly on the raw inputs and does not include multiple layers of feature extraction. Logistic regression is important to consider, however, because it is widely used and also acts as the final layer in many deep classification models.

The idea is to model the conditional probability \(P(y \mid x)\). Given a matrix of weights
Chapter 2. Background

\( W \in \mathbb{R}^{D \times C} \) and a vector of biases \( b \in \mathbb{R}^C \), the probability is defined as,

\[
\mathbf{a} = \mathbf{x}^\top \mathbf{W} + \mathbf{b},
\]

\[
P(y_c = 1 | \mathbf{x}) = \text{Softmax}(\mathbf{a})_c,
\]

\[
\text{Softmax}(\mathbf{a})_c = \frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a_{c'})}.
\]

The training objective of logistic regression is to maximize the log-probability, or equivalently to minimize the negative log-probability. This is known as the cross-entropy loss function. Given \( N \) examples, we assume that each example is an iid sample from \( P(y | x) \), therefore the objective becomes a sum over \( N \) terms.

\[
\ell(W, b) = \sum_{n=1}^{N} \sum_{c=1}^{C} \delta(y_{nc} = 1) \log P(y_{nc} = 1 | x).
\]

(2.1)

Where \( \delta(a, b) \) is the indicator function that is 1 if \( a = b \) and 0 otherwise. For convenience, we can use the average cross-entropy over the training set rather than the sum. This will allow us to use mini-batch stochastic gradient descent where the learning rate will not depend on the size of the mini-batch.

2.1.2 Multi-layer Perceptrons

Now that we have defined the output layer and the loss function, we can define the lower layers that produce features. A multilayer perceptron (MLP) [Rumelhart et al., 1985] is a simple feed-forward neural network that consists of several layers of matrix multiplication, each followed by an element-wise nonlinear function known as a nonlinearity. The layers below the final classification layer and above the input layer are known as hidden layers, and each dimension of a hidden layer is known as a hidden unit. With each progressive layer, the MLP is designed to tease out progressively more nonlinear functions of the input.

Intuitively, the lower layers learn to represent simple features, while the higher layers learn to compose these simple features into progressively more abstract concepts. In vision, the lower layers often learn Gabor-like filters, while the higher layers learn concepts such as faces, cars, etc., depending on the training data. Another way of thinking about this is that network learns to map the input into a new space that is approximately linearly separable.

More concretely, an MLP with \( L \) layers will have weight matrices and bias vectors
\[ \theta = \{(W_i, b_i)\}_{i=1}^{L}. \] For a nonlinearity \( h(\cdot) \), Algorithm 1 defines the feed-forward pass.

**Algorithm 1** The feedforward pass of a multilabel classification MLP.

1: function MLPFeedForward(Parameters \( \theta \), input vector \( x \))
2: \( h_0 = x \)
3: for \( i = 1, 2, \ldots, L - 1 \) do
4: \( a_i = h_{i-1}^\top W_i + b_i \)
5: \( h_i = h(a_i) \)
6: end for
7: \( a_L = h_{L-1}^\top W_L + b_L \)
8: \( h_L = \text{Softmax}(a_L) \quad \triangleright \text{Classification layer} \)
9: end function

The loss function in Equation 2.1 can then be applied to the network output layer. Common nonlinearities include sigmoid: \( h(x) = \frac{1}{1 + \exp(-x)} \), tanh: \( h(x) = \tanh(x) \) and rectified linear unit: \( h(x) = \max(0, x) \).

**2.1.3 Convolutional Neural Networks**

When the input data consists of images, we can apply a powerful form of neural network called a convolutional neural network (CNN) [Le Cun et al., 1990]. Unlike general inputs, images have spatial structure; pixels are arranged in a grid, and nearby pixels tend to be highly correlated. Also, basic patterns can appear anywhere in an image, and are often related to a canonical pattern via an affine transformation. For example, the location of a particular object in an image can be changed by shifting it by e.g., moving the camera.

CNNs take advantage of this structure by learning local feature detectors that operate on a small region of the image. By applying the same feature detector to every region of the image, the network can learn complex patterns while greatly reducing the number of free parameters compared to an MLP. CNNs also display a much greater degree of invariance, especially to spatial shifting, than MLPs. This is further enhanced by using pooling layers, as we will describe below.

Consider an input image \( I \) consisting of three input channels corresponding to the red, green, and blue intensities of each pixel (typically a value between 0 and 255). A CNN applies a weight filter \( W \) to \( I \) via the convolution operator \( I * W \). Suppose the input to a convolution layer is a tensor of size \( U \times V \times W \), and the layer applies \( K \) filters of size \( U \times R \times Q \), then the output will have size \( K \times (V - R + 1) \times (W - Q + 1) \). Each 2-D slice along the first axis of size \( (V - R + 1) \times (W - Q + 1) \) is called a filter map and each element of each filter map is a hidden unit. The hidden units maintain a spatial layout.
within each filter map. As before, a nonlinearity can be applied after the convolution to each hidden unit. Notice that with each progressively higher layer, the hidden units within that layer will be functions of progressively larger regions of the input image.

For a computational speedup, a stride parameter $S$ can be used, which specifies the amount by which each filter is shifted spatially when performing the convolution. For $S > 1$, this results in fewer matrix multiplications and a smaller number of output units. Applying a valid convolution means that the output filter map will necessarily possess a smaller number of spatial dimensions than the input. The number of spatial dimensions can be maintained or even increased by using zero-padding around the input of width $P$.

A common way to reduce the size of the filter maps and enhance shift invariance is to include pooling layers that combine hidden units within each filter map. These typically operate on local $M \times M$ regions of each filter map and output a single hidden unit. The pooling operation is then applied spatially in a similar fashion to a convolutional operation. An empirically successful pooling operation is max pooling, which simply takes the maximum output of the units within the pooling region.

A convolutional network is typically composed of successive layers consisting of a convolution, a nonlinearity, and a pooling operation. For example, a widely used architecture is the Oxford VGG network [Simonyan and Zisserman, 2015]. Beyond these basic operations, other operations such as local contrast normalization can also be applied [Jarrett et al., 2009]. Several fully-connected layers can then be applied before the final classification layer.

### 2.2 Training Deep Models

Training a neural network involves adjusting the weights in order to minimize a target loss function, e.g., cross-entropy. Perhaps the most heavily used technique in the current literature is to use some form of gradient descent. The process of computing the gradient of the loss with respect to the weights in a feed-forward neural network is known as backpropagation [Rumelhart et al., 1985], which is an application of the chain rule to neural networks. An example of gradient descent training for a neural network is given in Algorithm 2, where $\nabla \ell(\theta; \{x_n, y_n\}_{n=1}^N)$ denotes the gradient of the loss with respect to the weights and biases of the network. The learning rate $\eta$ determines how much the parameters will change at each step of gradient descent and can be considered a hyperparameter. Note that this is a very simple form of gradient descent and does not consider more advanced techniques such as a line search. It is, however, a useful starting point for neural network training.
Algorithm 2 Simple gradient Descent Training.

1: function GRADDESCENT(Loss function $\ell(\theta; \{x_n, y_n\}_{n=1}^N)$, initial parameters $\theta_0$, $N$ input/output pairs $\{x_n, y_n\}_{n=1}^N$, learning rate $\eta$.)
2:  for $i = 1, 2, \ldots$ until convergence do
3:     Compute $\nabla \ell(\theta_{i-1}; \{x_n, y_n\}_{n=1}^N)$ via backpropagation.
4:     $\theta_i \leftarrow \theta_{i-1} - \eta \nabla \ell(\theta_{i-1}; \{x_n, y_n\}_{n=1}^N)$
5:  end for
6: end function

Recently, there has been a proliferation of deep learning software frameworks that implement reverse-mode automatic differentiation (a generalization of backpropagation) [Bergstra et al., 2010, Bastien et al., 2012, Collobert et al., 2011, Abadi et al., 2015], making it possible to rapidly iterate on new deep models.

2.2.1 Stochastic Gradient Descent

An extremely common assumption in supervised learning is that the data is independently and identically distributed according to some underlying generating process. The effect of this assumption is that the loss function becomes a sum over independent terms,

$$\ell(\theta; \{x_n, y_n\}_{n=1}^N) = \frac{1}{N} \sum_{n=1}^N \ell_n(\theta; x_n, y_n).$$

The gradient of the loss can therefore also be expressed as a sum over independent terms,

$$\nabla \ell(\theta; \{x_n, y_n\}_{n=1}^N) = \frac{1}{N} \sum_{n=1}^N \nabla \ell_n(\theta; x_n, y_n).$$

The result is that when performing gradient descent, each step requires $O(N)$ computation, corresponding to $N$ feed-forward and backward passes, one for each training example. Under the iid assumption however, we can use a simple Monte Carlo estimate of the gradient:

$$\nabla \ell(\theta; \{x_n, y_n\}_{n=1}^N) \approx \frac{1}{M} \sum_{m=1}^M \nabla \ell_m(\theta; \tilde{x}_m, \tilde{y}_m). \quad (2.2)$$

Where $(\tilde{x}, \tilde{y})$ are randomly selected pairs of inputs and targets from the training data. $M$ is known as the batch size and the subset of data selected for a gradient estimate is
known as a mini-batch. Stochastic gradient descent proceeds as normal gradient descent, with the exception that the gradients are computed using the Monte Carlo estimate instead of the full gradient. This makes each evaluation cost $O(N/M)$ so that $\theta$ can be updated more frequently in a given amount of time. Stochastic gradient descent provides noisy gradient directions, so the number of required steps to convergence will be higher, however if the batch size is large enough, then noise can be made small enough that this is greatly offset by the increased speed of each step. Stochastic gradient descent is an incredibly important algorithm for neural networks as it allows them to be trained on massive datasets and this is the regime in which they tend to dominate other approaches for classification.

[Robbins and Monro, 1951] provided conditions on the learning rate $\eta$ for stochastic gradient descent to asymptotically converge to a stationary point. Essentially, we must define a learning rate schedule $\{\eta_i\}$ that satisfies the following conditions:

$$\sum_{i=1}^{\infty} \eta_i = \infty,$$

$$\sum_{i=1}^{\infty} \eta_i^2 < \infty.$$

One schedule that satisfies this condition is $\eta_i \propto \frac{1}{i}$. Practically however, a constant learning rate often works well, or a less aggressive schedule such as a linear decay. More recently, there has been a proliferation of techniques for adjusting the learning rate dynamically, such as AdaGrad [Duchi et al., 2011], RMSProp [Hinton, 2012], and ADAM [Kingma and Ba, 2015]. Another common trick is the use of momentum [Polyak, 1964] and Nesterov momentum [Nesterov, 1983, Sutskever et al., 2013] that act to smooth out the noise in the gradients over time by preventing the descent procedure from changing directions too rapidly. Finally, Polyak averaging [Polyak and Juditsky, 1992] can be used to average away the noise toward the end of training when the parameters are oscillating around a stationary point. All of these methods have specialized tuning parameters that can greatly influence the speed of neural network training and the quality of the final result.

### 2.2.2 Regularization

Neural networks are powerful models for function estimation, in particular they are universal approximators in the sense that a large enough network has the capacity to arbitrarily approximate any function [Sutskever and Hinton, 2008]. Practically, the result of
this is that neural networks are quite prone to overfitting. The solution to this problem is to find ways to reduce the capacity of the neural network, while still maintaining enough capacity to model the data. Below we outline several common methods for regularizing neural networks; note that this is not an exhaustive list, but rather contains some of the more common methods.

**Network size** One very simple technique is to directly reduce the capacity of the network by reducing the number of hidden units—either the number of layers, or the number of units in each layer.

While this approach is perhaps the most direct regularizer, it may be the case that classifier being learned still needs the flexibility of a large network, for example if the classification boundary is highly nonlinear. In this case, with a large network there could be many configurations that produce the same classifier, and these may be easier to discover through gradient descent than with a smaller network. Once a large network is learned, however, it has been shown that the function it implements can be transferred to a smaller network [Ba and Caruana, 2014]. This suggests that the issue with reducing the capacity may be an issue of learning rather than one of representation ability.

**Weight decay** Another way to restrict the capacity of the network is to add a penalty function to the learning objective that restricts the magnitude of the network weights.

\[
\ell(\theta) = \ell_{\text{obj}}(\theta) + \lambda \ell_{\text{reg}}(\theta). \tag{2.3}
\]

where \( \ell_{\text{obj}}(\theta) \) is e.g., cross entropy and \( \ell_{\text{reg}}(\theta) \) is usually chosen to be the L2 penalty,

\[
\ell_{\text{reg}}(\theta) = \sum_i \theta_i^2.
\]

Here, \( i \) indexes a particular weight. \( \lambda \) is a hyperparameter that governs the strength of the penalty. This has the effect of smoothing the output function.

An alternative, but equivalent form of weight decay is to constrain the magnitude of the square of each weight. Indeed, adding the penalty (2.3) gives the Lagrange dual of the constrained problem. A structured version of this that is sometimes used, is to constrain the norm of all of the weights to a hidden unit. This gives the individual weights more flexibility, while still limiting the capacity of the network.

**Dropout** Dropout [Srivastava et al., 2014] is a particular instance of adding noise to the network during the training process. When the network is overfitting, it may fixate
on a particular hypothesis that fits the data, but be very sensitive to perturbations of
the network parameters. Adding noise to the hidden units forces the network to focus
on robust solutions instead, which can help with generalization.

In the case of dropout specifically, the noise is applied by multiplying each hidden
unit with a realization of a Bernoulli random variable, effectively removing a random set
of hidden units from the network with each forward pass. This reduces the dependence
that hidden units have on each other since the accuracy of the network must be robust
to the absence of units. In dropout, we usually refer to the expected drop rate $p$, which
is 1 minus the Bernoulli parameter, also known as the keep rate.

**Early stopping** A very simple and effective form of regularization is to halt the learn-
ing process before the loss function on the training set has been fully minimized. Essentially, as the function is fitting the data there may come a point where it begins to overfit.
At this point, continuing to learn is actually harmful to generalization performance. Using a held-out validation set, the iteration in which this roughly starts to happen can be
detected and the learning can be stopped.

### 2.3 Collaborative Filtering

Collaborative filtering is a problem that emerged from the idea of building a recommender
system. The scenario is that there are $N$ users and $M$ items. Each user $i$ has rated some
subset $S_i$ of the items with a rating $R$ which could be, e.g., an integer score from 0 to 5.
Given this information, the goal is to build a model that could predict the rating user
$i$ gives to item $j$ so that the most appropriate items can be recommended to the user.
A concrete setting where this problem arises is in movie recommender systems where
the goal is to recommend movies for users to watch. There are many solutions to this
problem, but below we will outline a particularly popular approach, probabilistic matrix
factorization.

#### 2.3.1 Probabilistic Matrix Factorization

Probabilistic matrix factorization (PMF) [Salakhutdinov and Mnih, 2008] builds a prob-
abilistic model of the collaborative filtering problem whose parameters are estimated via
maximum likelihood or maximum a posteriori (MAP) training.

Suppose there is a true underlying ratings matrix $R$ and that we observe a subset
of these values. PMF approximates the true underlying matrix as $R \approx U^T V$, where $U$
and \( V \) are matrices of size \( K \times N \) and \( K \times M \) respectively. The idea is that the true underlying ratings matrix should exhibit low rank structure, which translates to the idea that there are sets of similar users who will give similar ratings to each movie.

For user \( i \) and movie \( j \), PMF defines the following distribution over the ratings matrix,

\[
P(R | U, V) = \prod_{i=1}^{N} \prod_{j=1}^{M} \mathcal{N}(R_{ij} | U_i^T V_j, \sigma^2)^{O_{ij}},
\]

\[
P(U) = \mathcal{N}(U | 0, \sigma_U^2 I),
\]

\[
P(V) = \mathcal{N}(V | 0, \sigma_V^2 I).
\]

Where \( O_{ij} \) is 1 if user \( i \) has rated movie \( j \) and 0 otherwise, and \( \sigma^2, \sigma_U^2 \) and \( \sigma_V^2 \) are the observation noise and prior variances respectively. If we treat the noise and prior variances as hyperparameters, then the MAP training loss becomes:

\[
\ell(U, V) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{M} O_{ij} (R_{ij} - U_i^T V_j)^2 + \frac{\lambda_U}{2} \sum_{i=1}^{N} \|U_i\|^2 + \frac{\lambda_V}{2} \sum_{j=1}^{M} \|V_j\|^2
\]

Where \( \lambda_U = \frac{\sigma^2}{\sigma_U^2} \) and \( \lambda_V = \frac{\sigma^2}{\sigma_V^2} \). For convenience, we can divide the objective by \( \sum_{i=1}^{N} \sum_{j=1}^{M} O_{ij} \) and train this objective by stochastic gradient descent, randomly picking sets of users and items.

### 2.4 Topic Modelling

It is natural when reading an article to describe the article in terms of high level topics. For example, a single news article might be about the topics economics, politics, and technology. In machine learning, topic modelling is the problem of describing a document in terms of these kinds of topics. In the standard setup, a topic is defined to be a frequently co-occurring set of words. The model is trained in an unsupervised manner, and the topics are usually assigned meaningful labels post-hoc by inspecting the resulting word combinations. Specifically, we assume that we have a corpus of documents \( D_1, D_2, \ldots, D_M \) and a vocabulary of words \( w \in \mathcal{W} \) where \( \mathcal{W} \) is the set of elementary vectors of size \( N \), where \( w_i = 1 \) if \( w \) is the \( i \)th word in the vocabulary. In the basic setup, each document is represented of a set of words with their associated counts of how many times they appear in the document.


2.4.1 Online Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) [Blei et al., 2003] is a particular Bayesian topic model. We assume that there are $K$ topics and that topics are defined as a multinomial distribution over words, represented by a $K \times N$ matrix $\beta$. LDA assumes a document $d$ is generated from the following process,

1. Draw the number of words $N_d$ from Poisson($\lambda$).
2. Draw a distribution over topics $\theta_d$ from Dirichlet($\alpha$).
3. For each word 1 to $N_d$ in the document,
   (a) Draw a topic $z_j$ from Multinomial($\theta_d$),
   (b) Draw a word from the multinomial in the $z_j$th row of $\beta$.

If we marginalize over the topic assignments, we get that the distribution of word $i$ in document $d$, $w_{id}$ is represented as $P(w_{id}|\theta_d, \beta) = \theta_d^\top \beta_i$ [Hoffman et al., 2010]. In other words, LDA can be seen as a generalization of PMF to the case where the ratings matrix consists of counts and a multinomial likelihood is used instead of a Gaussian.

LDA is typically fit via Bayesian inference as opposed to maximum likelihood. The goal then is to infer the posterior distribution over the parameters $\beta$, $\theta$, and the topic assignments $z$, $P(\beta, \theta, z|D_1, D_2, \ldots, D_M, \lambda, \alpha)$. One effective approach is to use variational inference, where the approximating distribution assumes independence between $\beta$, $\theta$, and $z$. The main issue with the standard Bayesian approach is that it does not scale gracefully to large numbers of documents. In [Hoffman et al., 2010], this was remedied with a method called online LDA using stochastic variational inference [Hoffman et al., 2013].

2.5 Types of Hyperparameters

One thing that each of the above models have in common is that they have tunable hyperparameters that are tuned separately from model parameters. These include architectural parameters that govern the structure of the model, such as the number of layers and hidden units per layer in an MLP, the filter size of a CNN, the nonlinearities and pooling types, the number of factors in PMF and the number of topics in LDA. There are the hyperparameters that directly govern regularization, such as the weight decay strength $\lambda$, the drop rate $p$, and the early stopping iteration. Then there are optimization hyperparameters such as the learning rate and momentum. One can step back and
make the choice of model or optimization algorithm itself a hyperparameter. One can even include hyperparameters for data pre-processing, such as whether or not to whiten the data.

Essentially, the number and types of hyperparameters is only constrained by the imagination. However, as the number of choices increase, the space of possibilities becomes exponentially larger and the search becomes more difficult. We will therefore assume that the user has selected an appropriate model that they would like to tune, the optimizer they would like to use, and any preprocessing, and focus on automating the tuning of the remaining free hyperparameters.

2.6 Bayesian Optimization

Bayesian optimization [Jones et al., 1998, Lizotte, 2008, Brochu et al., 2010b, Snoek et al., 2012, Bergstra et al., 2011, Osborne et al., 2009] is a framework for solving the following constrained global optimization problem:

$$\min_{x} f(x)$$

subject to $x \in \mathcal{X}$.

Where $f$ is an unknown, possibly noisy function and $\mathcal{X}$ is a $D$-dimensional compact set. That is, we can only gather information about $f$ by querying it directly, yielding potentially noisy observations $y$. In particular, we also assume that querying $f$ is an expensive operation, so a good optimizer is one that minimizes $f$ in as few evaluations as possible. Bayesian optimization is closely related to the design of experiments, so we will use the term experiment as a synonym for a query. When $\mathcal{X} \subset \mathbb{R}^D$, the set will usually be subject to box constraints, so we can equivalently reparameterize $\mathcal{X}$ so that the problem lies in the unit hypercube.

The main idea behind Bayesian optimization is that we will use samples of the true function in order to construct an estimate of the function known as a surrogate. This surrogate can be used as a proxy for the true function and queried liberally in order to select a high quality candidate for the next experiment. In estimating $f$ using a finite number of samples, there might be an infinite number of possible functions that could be generated that are consistent with the data. Therefore, we impose a prior over how we think the function behaves, e.g., we might a-priori believe that it is smooth and twice differentiable. As we collect data, we update this distribution to form a posterior over possible functions that are consistent with the observed data.
Put another way, we need a model that we can update when new data is observed, and that is capable of producing reasonable uncertainty estimates. Although there are many possible choices, in this thesis we will focus on using Gaussian processes [Rasmussen and Williams, 2006] as the surrogate model.

Intuitively, the uncertainty will be highest in areas that we have not yet explored. This gives us a principled way to trade off exploration—searching areas of the function where we have very little information—with exploitation—querying the function in areas we believe will yield better values. This trade-off is crucial in terms of attempting to globally minimize the function efficiently, without getting stuck in a local minimum. To do this, we use an acquisition function, which can be thought of as a utility function that describes this trade-off.

Given an objective function $f$, a model $g$, and an acquisition function $a$, we can write out the full Bayesian optimization algorithm, shown in Algorithm 3.

**Algorithm 3** The Bayesian optimization algorithm.

1: function $\text{BayesOpt}(\text{Function } f, \text{ Model } g, \text{ acquisition function } a, \text{ initial set of observations } D_0)$
2:     for $n = 1, 2, \ldots$ do
3:         Find $x_n = \arg\max_x a(x; g, D_{n-1})$
4:         Evaluate $f(x_n)$ to generate observation $y_n$
5:         Update $D_n = D_{n-1} \cup (x_n, y_n)$
6:         Update the model $g$ using $D_n$
7:     end for
8: end function

In the next few sections, we will go into more detail about the different aspects of the Bayesian optimization framework.

### 2.6.1 Gaussian Processes

Gaussian processes are a popular class of Bayesian models for regression and classification. Their popularity stems from the fact that they are flexible, easy to implement, and difficult to overfit. For Bayesian optimization, we will further rely on the convenient property that they give analytic uncertainty estimates along with their predictions.

Informally speaking, a Gaussian process (GP) [Rasmussen and Williams, 2006] defines a distribution over functions. That is, a draw from a Gaussian process is an entire function over some domain $\mathcal{X}$. It is specified by a mean function $m : \mathcal{X} \to \mathbb{R}$ and a covariance function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Usually, the covariance function is chosen to be
a Mercer kernel [Shawe-Taylor and Cristianini, 2004], meaning it produces a positive semidefinite matrix for any set of inputs, although this is not always the case [Rasmussen and Williams, 2006, Chapter 4].

Suppose we have a dataset consisting of $N$ input-output pairs $\mathcal{D}_N = \{(x_i, y_i)\}_{i=1}^N$. For convenience, let $X$ denote the $N \times D$-sized data matrix formed by arranging the inputs in rows. Let $y \in \mathbb{R}^N$ denote the vector of observed values. A Gaussian process regression model assumes the following generative procedure:

$$
\begin{align*}
  f & \sim \mathcal{N}(m, K_{XX}), \\
  y & \sim \mathcal{N}(f, \sigma^2 I).
\end{align*}
$$

(2.4)

Where $K_{XX}$ is the Gram matrix, the matrix of kernel evaluations between all training points in $X$, $I$ is the identity matrix, and $m$ is the vector of mean function values evaluated at all training points. This says that we draw some latent function values from a multivariate Gaussian distribution, and then add independent Gaussian noise to generate the observed variables. In this case, the Gaussian process acts as a prior on the latent function values $f$.

Note that the kernel and mean functions are usually parameterized. We will call the set of these, as well as the observation noise $\sigma^2$, $\Theta$, or the Gaussian process hyperparameters (not to be confused with the hyperparameters we’re optimizing with Bayesian optimization). For brevity, we will not refer to these explicitly in the Gaussian process formulas unless needed.

The basic premise of a Gaussian process is that similar inputs should yield similar observations. It is the kernel function that defines what we mean by similar. For example, a commonly used kernel is the squared exponential (SE) kernel (also known as a Gaussian, or radial basis function kernel),

$$
\begin{align*}
  k_{SE}(x, x') &= \theta_1 \exp \left( -\frac{\|x - x'|^2}{\theta_2^2} \right),
\end{align*}
$$

(2.5)

Where $\|\cdot\|$ is the $\ell_2$ vector norm. Here, points are considered similar if they have a small Euclidean distance. $\theta_1$ and $\theta_2$ represent the amplitude and length scale parameters of the kernel. The amplitude affects the magnitude over which the function values vary, and the length scale determines the effective neighborhood of the function, or the distance over which points can meaningfully influence each other.

The SE kernel is a good example of how we can impose priors on functions with
a Gaussian process. Samples from a GP using this covariance produces very smooth, infinitely differentiable functions [Rasmussen and Williams, 2006]. Another popular covariance function is the Matérn-$\frac{5}{2}$ kernel, which produces a prior over twice differentiable functions.

$$k_M(x, x') = \theta_1 \left( 1 + \sqrt{\frac{5}{\theta_2}} \frac{\|x - x'\|}{\|x - x'\|^2} + \frac{5}{3\theta_2^2} \right) \exp \left( -\sqrt{\frac{5}{\theta_2}} \frac{\|x - x'\|}{\|x - x'\|^2} \right). \quad (2.6)$$

Examples of the SE and Matérn-$\frac{5}{2}$ kernel are shown in Figures 2.1a and 2.1b, along with the effect of varying the length scales. In the case of multi-dimensional data, each dimension can have its own length scale. The kernel then becomes an automatic relevance determination, or ARD kernel, as opposed to an isotropic kernel, where every dimension shares the same length scale. The term ARD is used because when learning the kernel hyperparameters, the GP can learn the relative importance of each dimension by adjusting its length scale accordingly. Dimensions with large length scales will have relatively little influence, since changes along that dimension will not have significant changes on the function.

A very useful kernel is the noise kernel. The noise kernel adds independent Gaussian noise to each observation. This is defined as

$$k_N(x, x') = \sigma^2 \delta(x, x'),$$

where $\sigma^2$ is known as the nugget, or the variance of the noise, and $\delta(a, b) = 1$ if $a_i = b_i \forall i$. We can define a GP kernel as the sum of two kernels such as a Matérn and a noise kernel, since the sum of two kernels is also a valid kernel. Using a noise kernel is mathematically equivalent to having a noisy Gaussian likelihood for the observations, as in Equation (2.4). An example of the noise kernel added to the Matérn-$\frac{5}{2}$ kernel is shown in Figure 2.1c.

Finally, if we would like the noise to be correlated by distance, rather than independent, a good choice is the Ornstein-Uhlenbeck kernel,

$$k_{OU}(x, x') = \theta_1 \exp \left( -\frac{\|x - x'\|}{\theta_2} \right).$$

This kernel describes Brownian motion and corresponds to a Matérn-$\frac{1}{2}$ kernel. It produces functions that are continuous, but not differentiable anywhere. We show draws from this kernel with different length scales in Figure 2.1d.

GPs are appealing models for regression because they are extremely flexible and
Figure 2.1: Functions drawn from a (a): squared exponential, (b) Matèrn-$\nu_2$, (c): Matèrn-$\nu_2$ plus noise and (d): Ornstein-Uhlenbeck kernel. Each color is drawn using the same seed in order to highlight the differences between the kernels. Different colors correspond to different length scales. We use $\theta_1 = 1$ for all kernels. Best viewed in colour.
have a simple closed-form posterior predictive distribution. Suppose we would like to make a prediction at a test point. Using the marginalization and conditioning rules of the Gaussian distribution given in Appendix A.2, it can be shown that the posterior predictive distribution takes the following form,

\[
P(f|\mathcal{D}_N, \mathbf{x}) = \mathcal{N}(f; \mu(\mathbf{x}), v(\mathbf{x})) ,
\]

\[
\mu(\mathbf{x}) = \mathbf{K}_x\mathbf{x} (\mathbf{K}_{xx} + \sigma^2 \mathbf{I})^{-1} (\mathbf{y} - m(\mathbf{x})) ,
\]

\[
v(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{K}_{xx} (\mathbf{K}_{xx} + \sigma^2 \mathbf{I})^{-1} \mathbf{K}_{xx}^\top .
\]

Where \( \mathbf{K}_{xx} \) is a \( 1 \times N \) vector of the kernel function between \( \mathbf{x} \) and each point in \( \mathbf{X} \).

The most computationally expensive part of GP inference involves inverting the Gram matrix, which takes \( O(N^3) \) time. Assuming this step has been pre-computed, making predictions at a new test point requires matrix-vector products, which takes \( O(N^2) \) time. This makes ordinary Gaussian processes intractable for large-scale problems, however in Bayesian optimization we will typically be working in the low-data regime where the GP computations won’t be too cumbersome.

### 2.6.2 Acquisition Functions

Once a model has been specified and trained on the data we have observed so far, the next step is to use the model to choose the next experiment to run. The goal of Bayesian optimization is to find the globally optimal solution in as few function evaluations as possible. To do this we use an acquisition function \( a(\mathbf{x}) \), a utility function that measures the predicted quality of a candidate experiment. There have been many proposals for acquisition functions based on different principles, and we will explore some of the more common choices below. We will focus on maximizing \( a(\mathbf{x}) \) in order to maintain the idea that it acts as a utility function.

The proper choice of acquisition functions depends on a number of factors. For example, if we only care about finding the globally optimal solution, and we have an infinite amount of time to do so, then we could perform an exhaustive search. Alternatively, if our budget does not allow for any more experiments, then we will have to suggest our best guess at the current point in time. Usually we have some finite budget. We could then try to use this budget in an acquisition function and try to find the best point to query, assuming we will query \( t \) points in the future. As \( t \) grows however, this can become computationally intensive as we would have to potentially forecast all outcomes of all ways of choosing \( t \) different points. Therefore, most commonly used acquisition func-
tions are myopic in that they assume that we are allowed to query one more point before terminating the search (even if this is not necessarily true). This provides a balance between the efficiency of computation and optimization.

2.6.2.1 Pure Exploration and Exploitation

Two simple heuristics are to maximize $\sigma(x)$ or $-\mu(x)$ (since we are minimizing the unknown function). Maximizing $\sigma(x)$ would encourage queries away from previously observed locations and would therefore do a good job of exploring the space. The issue with this approach is that it will continue exploring indefinitely, long after the model is confident in the location of the global minimum. As for maximizing $-\mu(x)$, this runs a serious risk of getting trapped in what might not even be a local optimum [Jones et al., 1998] since there is no incentive to explore in areas away from the current best solution.

2.6.2.2 Probability of Improvement

An alternative way to define the quality of a point is whether we think that the function at that point will improve upon our current best observation. Thus, [Kushner, 1964] proposed the probability of improvement (PI) heuristic. Let the current best observation be $y_*$ at point $x_*$. For a Gaussian process model, PI can be computed in closed form as:

$$a_{PI}(x) = P(f(x) \leq y_*)$$

$$= \Phi(\gamma(x)),$$

$$\gamma(x) = \frac{y_* - \mu(x)}{\sigma(x)}.$$

Where $\Phi(z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{1}{2}z^2 \right) dz$ is the standard Gaussian cumulative distribution function.

The principal issue with PI is that it can at times be extremely greedy. If the search falls into a particularly low basin, for example, then exploring an infinitesimal distance from the current best point can yield a high PI, therefore the search will simply continue to explore near $x_*$ rather than exploring points with higher uncertainty.

To combat this, [Kushner, 1964] proposed adding a parameter $\xi \in \mathbb{R}^+$ in order to
encourage greater exploration,

\[
a_{\text{PI}}(x) = P(f(x) \leq y_* - \xi),
\]

\[
= \Phi(\gamma(x; \xi)),
\]

\[
\gamma(x; \xi) = \frac{y_* - \xi - \mu(x)}{\sigma(x)}.
\]

\(\xi\) must be set appropriately in order to provide the right balance of exploration and exploitation. Currently, this seems to be an empirical question and the solution likely varies from problem to problem.

### 2.6.2.3 Expected Improvement

The reason for the pathological behaviour of PI is that it focuses on whether there will be an improvement, but not on the magnitude of the improvement. One way to correct this is to look at the expected improvement (EI), as proposed in [Močkus, 1975].

\[
a_{\text{EI}}(x) = \mathbb{E}_{P(f|D_N, x)}[\max(y_* - f, 0)]
\]

\[
= \int_{-\infty}^{\infty} \max(y_* - f, 0)P(f|D_N, x)df,
\]

\[
= \int_{-\infty}^{\infty} \max(y_* - f, 0)\mathcal{N}(f|\mu(x), \sigma^2(x)) df,
\]

\[
= \sigma(x) (\gamma(x)a_{\text{PI}}(x) + \mathcal{N}(\gamma(x); 0, 1)).
\] (2.9)

Where \(\max(y_* - f, 0)\) is known as the improvement function.

EI has been empirically demonstrated to be an effective acquisition function [Snoek et al., 2012]. Rates of convergence for EI have been given in [Bull, 2011], and non-myopic extensions have been derived [Ginsbourger et al., 2010].

### 2.6.2.4 Entropy Search

One view of Bayesian optimization is that at each step we are gathering information about where the minimum of the function might be. The entropy search (ES) strategy [Hennig and Schuler, 2012] quantifies this by considering the posterior distribution over the location of the minimum \(P_*(x|D_N)\). For a discrete set of \(M\) points, we can write \(P_*(x|D_N)\) as:
Chapter 2. Background

\[ P_\star(x_i|D_N, x_1, \ldots, x_M) = \int_{y_1, \ldots, y_M} P(y_1, \ldots, y_M|x_1, \ldots, x_M, D_N)|y_i < y_{-i}|dy, \]  

(2.10)

where \( y_{-i} \) refers to the values of all \( M \) points except for \( i \), and \([\cdot]\) refers to the indicator function, which evaluates to 1 if the condition within is true.

This distribution is captured implicitly by modelling the distribution over function values. The idea behind entropy search is to choose the point that most reduces our uncertainty over the location of the minimum.

\[ a_{ES}(x) = H(P_\star(x'|D_N)) - \mathbb{E}_{P(y|D_N, x)}[H(P_\star(x'|D_N \cup \{x, y\})]. \]  

(2.11)

Where \( P(y|D_N, x) \) is the probability of observing \( y \) at \( x \) under our model and \( H(P) = -\int P(x) \log P(x)dx \) is the differential entropy of distribution \( P \). In other words, entropy search seeks to choose the point \( x \) that maximizes the information gain in terms of the distribution over the location of the minimum. It does this by first computing the entropy of \( P_\star(x'|D_N) \) given the current observations (the left term), and then computing the expected entropy after incorporating \( x \) into the set of observations (the right term).

For continuous spaces, computing the entropy can be difficult. One simple strategy is to only consider the entropy over a discrete grid of points. A useful heuristic is to choose this grid to be local maximizers of expected improvement since these points are likely to contain information about the location of the minimum of the function, especially towards the end of the search.

Two ways that have been proposed in the literature to compute the entropy and expected entropy are Monte Carlo simulation and expectation propagation (EP) [Hennig and Schuler, 2012]. The Monte Carlo method can have high variance and can only be computed on a discrete set of points, while the EP method scales as \( O(N^4) \), making it potentially very expensive.

Nevertheless, it can be shown that ES tends to promote exploration more than EI and so it can be useful in problem instances where EI behaves too greedily [Hernández-Lobato et al., 2014].
2.6.3 Practical Considerations

2.6.3.1 Learning Gaussian Process Hyperparameters

The parameters of the kernel and mean functions, and the observation noise can be trained by maximizing the log-marginal likelihood of the observations \( \log P(y|X) \) (the latent function values are marginalized), also known as the model evidence in Bayesian statistics.

\[
\ell(\Theta) = -\frac{1}{2} y^\top (K_{xx} + \sigma^2 I)^{-1} y - \frac{1}{2} \log |K_{xx} + \sigma^2 I| - \frac{N}{2} \log(2\pi).
\]

This is known as maximum marginal likelihood [Bishop, 1995], since it involves performing maximum likelihood after marginalizing out a set of latent variables. A prior \( P(\Theta) \) can also be specified, in which case maximizing the log-likelihood plus the prior is known as empirical Bayes [Robbins, 1964]. Alternatively, we can account for uncertainty in the GP hyperparameters by providing a fully Bayesian treatment. This is an effective technique when the number of observations is small since it helps prevent over-fitting. Practically, this can be done by drawing samples from the posterior distribution using Markov-chain Monte Carlo (MCMC) [Neal, 1993]. In this thesis we will use slice sampling [Neal, 2003, Murray and Adams, 2010] as a black-box method for drawing approximate posterior samples because of its ease of implementation and lack of tuning parameters.

2.6.3.2 Initializing the Search

In the early stages of the search, the model generally has so little data that its estimates are not very reliable. It can be beneficial in this case to initialize the search with a few exploratory steps, called the initial design. Random search [Bergstra and Bengio, 2012] is one possible way to generate an initial design. Another popular strategy is to use a quasi-random low discrepancy sequence, such as a Sobol sequence.

2.6.3.3 Optimizing the Acquisition Function

So far we have discussed various acquisition functions that are typically used in Bayesian optimization, however each of these strategies themselves must be optimized in order to select a new experiment. In fact, theoretical convergence results currently requires that these acquisition functions can be globally optimized [Bull, 2011]. Since these acquisition functions yield non-convex programs, this can be a difficult task. In practice, Bayesian optimization systems typically use some form of local search [Hutter et al., 2010] or other
global optimization techniques, including CMA-ES [Hansen, 2006, Bergstra et al., 2011],
divided rectangles [Jones et al., 1993, Brochu et al., 2010b], or random search combined
with quasi-Newton refinement [Snoek et al., 2012]. We use the latter strategy in our
experiments.

2.6.3.4 Integrated Acquisition Functions
When using a fully Bayesian treatment of GPs, the GP will no longer provide a pointwise
mean and variance prediction, but rather a distribution over these quantities. In this case,
the proper way to incorporate uncertainty in the GP parameters into the acquisition
function is to compute the expected acquisition under this posterior distribution. This
is known as the integrated acquisition function,

\[ a_{\text{int}}(x) = \int_{\Theta} a(x; \Theta)P(\Theta|\mathcal{D})d\Theta. \]

When using MCMC sampling, we will approximate the integrated acquisition function
with an average over GP hyperparameter samples.

\[ a_{\text{int}}(x) \approx \frac{1}{S} \sum_{s=1}^{S} a(x; \Theta_s), \]
\[ \Theta_s \sim P(\Theta|\mathcal{D}). \]

Where \( \Theta_s \) are approximate samples from \( P(\Theta|\mathcal{D}) \). From this point forward we will
implicitly assume that the integrated acquisition function is being used when the GP
hyperparameters are inferred using MCMC, unless otherwise noted.

2.6.4 Extensions
2.6.4.1 Constraints
Bayesian optimization is based around the idea that after running an experiment, we will
observe some score that determines how well the experiment went. When tuning machine
learning models, however, we can run into situations where the score will be arbitrarily
bad or indeterminate. For example, when tuning a neural network certain architectures
may not fit in memory, or if the learning rate is too high then the training procedure
might diverge. If we simply ignore these experiments then the uncertainty in these areas
will remain high and the acquisition function will continue to recommend these points.
One solution is to assign a fixed, high value to these points in order to discourage the search from returning to these areas. This may perhaps work for a random forest, but will violate the smoothness assumption inherent in GP models.

Recent work has instead focused on modifying Bayesian optimization to handle these a-priori unknown constraints [Bernardo et al., 2011, Gelbart et al., 2014]. They are called unknown because we do not know their functional form in advance. The idea is that, much like the response surface approach of Bayesian optimization, we also try to model the constraint surface. We can use the objective model to probe the function, and use the constraint model to avoid areas that are likely to violate any constraints.

Let $\mathcal{R}$ denote regression and $\mathcal{C}$ denote classification. In addition to the regression data we normally collect, denoted here as $\mathcal{D}_N^R = \{(x_n, y_n)\}_{n=1}^N$, we now collect additional data $\mathcal{D}_N^C = \{(x_n, c_n)\}_{n=1}^N$ where $c_i \in \{0, 1\}$. Here, $c_i = 1$ means that an experiment successfully returned a $y_i \in \mathbb{R}$, while $c_i = 0$ means the experiment violated a constraint and hence did not return any value.

In [Gelbart et al., 2014], a Gaussian process classifier (GPC) [Neal, 1998] is used as the constraint model. A Gaussian process classifier is the same as a GP, except that a Bernoulli observation model is used instead of a Gaussian.

$$P_C(c|X_C, \Omega) = \int \left( \prod_{n=1}^{N_c} P(c_n|f_n) \right) P(f|X_C, \Omega) df,$$

$$P(c_n|f_n) = \Phi(f_n)^{c_n} (1 - \Phi(f_n))^{1-c_n}.$$

Where $\Phi(\cdot)$ is the probit function (a sigmoid could be used instead), and $P(f|X_C, \Omega)$ is a Gaussian process prior over function values at the data points in $X_C$ with kernel hyperparameters $\Omega$.

The posterior over function values $P_C(f|\mathcal{D}_N^C, \Omega)$ can be derived via Bayes rule, however since the observation model is not Gaussian it is intractable. Therefore, approximation techniques such as the Laplace approximation [Williams and Barber, 1998], variational inference [Gibbs and MacKay, 2000], expectation propagation [Minka, 2001], and MCMC using e.g., elliptical slice sampling [Murray et al., 2010] are often used. The GPC is a useful model because like the GP, it captures uncertainty over predictions and therefore will not be overconfident in regions of the search space that have not been explored.

In [Gelbart et al., 2014], expected improvement with a GP is multiplied by the prediction of a GPC to form a constrained acquisition function:
\[ a\text{EIC}(x) = a\text{EI}(x)P_C(c = 1|x, \mathcal{D}_N^C, \Omega). \]

This will yield the usual EI when the GPC is confident that a point is valid, and no improvement when the GPC is confident that a constraint will be violated at that point.

### 2.6.4.2 Parallelism

A highly effective way to speed up Bayesian optimization is through parallelism. If we have the resources to run several experiments in parallel, then we can cover the space much more effectively in a given amount of time. Naive algorithms like grid search and random search are trivially parallelizable, however Bayesian optimization is an inherently sequential algorithm, so parallelization requires careful thought.

In the parallel scenario, we assume that there are currently \( M \) experiments running at locations \( \{x^{(p)}_m\}_{m=1}^M \) that we call pending experiments. These have not yet returned a value. We would now like to suggest a new point \( x^{(p)}_{M+1} \).

In [Ginsbourger et al., 2010], several simple algorithms are proposed. The main idea behind these is that we will create some pseudo-observations \( \hat{y} \) at the pending locations. Letting \( \mathcal{D}_p = \{(x^{(p)}_m, \hat{y}_m)\}_{m=1}^M \) be the dataset of pending inputs/pseudo-observations respectively, we will augment the current dataset \( \mathcal{D}_N \) with this new data and then use this dataset to suggest a new point. The purpose of this is to promote exploration away from pending locations. The difference between these approaches lies in how \( \hat{y} \) is determined. One strategy is the constant liar, in which every \( \hat{y} \) is set to a pre-determined constant \( L \). A more intelligent approach is the Kriging believer strategy, where \( \hat{y}_m = \mu(x; \mathcal{D}_N) \), i.e., each pseudo-observation is set to the GP posterior predictive mean. Finally, [Snoek et al., 2012] simulates \( \hat{y} \) from the posterior distribution and averages over several simulated datasets. Unlike the Kriging believer, this accounts for uncertainty in the predictions so that the predictive variance does not collapse at the pending locations.
Chapter 3

Multi-Task

3.1 Introduction

Making deep learning models work properly is as much an art as it is a science. Beyond designing the appropriate neural network for a particular problem, a practitioner must apply the right set of architectural, optimization, data pre-processing and regularization techniques in order to ensure that the network learns properly. Much of this comes from experience and intuition, which is built over a long period of trial-and-error with different models on different problems. This ability to apply the knowledge learned on one problem to help solve another problem is something that people do extremely well, and it allows us to very quickly find workable models in a reasonable amount of time.

This chapter explores techniques for augmenting Bayesian optimization with the ability to transfer information between problems. To that end, we introduce multi-task Bayesian optimization. The basis for the idea is to apply multi-task Gaussian process models to the Bayesian optimization framework. By treating new problems as new tasks, we can adaptively learn the degree of correlation between problems and use this information to more quickly hone the search.

We use this new framework to solve several common issues found in automated hyperparameter optimization. One issue is the so-called “cold start” problem. The optimization must be carried out from scratch each time a model is applied to new data. If a model will be applied to many different datasets, or even just a few extremely large datasets, then there may be a significant overhead to re-exploring the same hyperparameter space. By using information learned from applying the model to old problems, and using the prior intuition that the same model applied to similar datasets will yield similar hyperparameter-performance relationships, we can rapidly make progress with new datasets. Furthermore, for large datasets one could imagine exploring a wide range
of hyperparameters on a small subset of data, and then using this knowledge to quickly find an effective setting on the full dataset with just a few function evaluations.

We demonstrate the utility of this approach in a number of different settings: using previous optimization runs to bootstrap new ones, optimizing multiple tasks simultaneously when the goal is maximizing average performance, and utilizing a small version of a dataset to explore hyperparameter settings for the full dataset. Our approach is fully automatic, requires minimal human intervention and yields substantial improvements in terms of the speed of optimization.

3.2 Multi-Task Gaussian Processes

In the field of geostatistics [Journel and Huijbregts, 1978, Goovaerts, 1997], and more recently in the field of machine learning [Seeger et al., 2005, Bonilla et al., 2007, Alvarez and Lawrence, 2011], Gaussian processes have been extended to work with vector-valued functions, i.e., \( f : \mathcal{X} \rightarrow \mathbb{R}^J \). We can interpret the \( J \) outputs of such functions as belonging to different regression tasks. The key to modeling such functions with Gaussian processes is to assume that all observations for all tasks are jointly Gaussian, and to define a useful covariance function \( k((x, j), (x', j')) \) between input-task pairs. One simple approach is called the intrinsic model of coregionalization, which transforms a latent function to produce each output. Formally,

\[
k_{\text{IMC}}((x, j), (x', j')) = k_X(x, x')k_J(j, j'),
\]

where \( k_X \) measures the relationship between inputs and \( k_J \) measures the relationship between tasks.

When every task is observed for every observation, we can write the Gram matrix as,

\[
K_{(x,j)(x,j)} = K_{XX} \otimes K_{JJ},
\]

where \( \otimes \) is the Kronecker product, \( K_{XX} \) is the input covariance and \( K_{JJ} \) is the task covariance. In this case we can exploit the Kronecker structure so that the cost of inversion is just the cost of inverting the input and task Gram matrices, \( O(N^3 + J^3) \), rather than \( O(N^3J^3) \). When some observations are missing, as will be typical in our case, the Gram matrix will not have any exploitable structure, so the cost of inversion will be cubic in the total number of observed values.

Along with the other kernel parameters, we can infer the parameters of \( k_J \) using evidence maximization or MCMC. To do this, we need to parameterize the task matrix
such that it is always positive definite. In our experiments, we parameterize the matrix by its Cholesky factor. We further constrain each element of the Cholesky to be non-negative, by storing the logarithm of each parameter, which imposes a prior that the tasks are either positively correlated or uncorrelated, but not negatively correlated. This can be expressed as,

\[
k_{ij}(j, j') = LL^\top,
\]

\[
L_{ij} = \begin{cases}  
\exp(\rho_{ij}) & \text{if } i \leq j, \\
0 & \text{otherwise.}
\end{cases}
\]

Where \(\rho_{ij}\) is an unconstrained, learned parameter. In our case, this is reasonable since we will choose tasks that we expect to be correlated to some degree. There are many other possible parameterizations, some of which are outlined in [Pinheiro and Bates, 1996].

### 3.3 Transferring Knowledge to a New Task

Using multi-task GPs we can take data from old tasks and combine it with data from new tasks, assigning task labels to each experiment accordingly. Under this model, the old data from previous tasks will act as pseudo-observations for the new task. To optimize the new task, we simply fix the task label to the new task and maximize expected improvement over \(X\).

An illustration of a multi-task GP versus a single-task GP and its effect on the expected improvement acquisition function is shown in Figure 3.1. Figure 3.1a shows three sample tasks generated from a multi-task GP. Figure 3.1b shows the predictions on the third task made by an independent GP when many observations are missing and Figure 3.1c shows the predictions made by a multi-task GP. The points represent observations, while the dashed line represents the predictive mean. The goal is to minimize the function over the third task. The curve shown on the bottom represents the expected improvement for each input location on this task. The independent GP fails to adequately represent the function and optimizing EI leads to a spurious evaluation. The multi-task GP utilizes the other tasks and the maximal EI point is much closer to the true minimum.
3.4 Optimizing an Average Function over Multiple Tasks

In many settings we have multiple objectives that we would like to optimize, often reflecting different trade-offs such as cost vs performance. Optimization over multiple objectives is known as multi-objective optimization and there are many strategies for solving this problem. One such strategy is to take a linear combination of these objectives, where the linear weights reflect the importance of each objective. This transforms a multi-objective problem into a single-objective problem [Boyd and Vandenberghe, 2004]. A particularly simple case of a linear combination is the arithmetic mean.

\[ \bar{f}(x) = \frac{1}{J} \sum_{j=1}^{J} f_j(x). \]
In practice we will not know \( f_j(x) \) directly, but will observe noisy realizations,

\[
\bar{y} = \frac{1}{J} \sum_{j=1}^{J} y_j.
\]

In some cases there may be a non-trivial cost associated with running each objective. In these cases, we can exploit the fact that the separate objectives are likely to be correlated and consider a more efficient strategy of only running a subset of objectives with each experiment. We use a multi-task GP to infer the remaining objectives, thereby giving us an estimate of the combined objective. This strategy allows us to run many more experiments in a fixed amount of time.

We motivate this approach by considering a finer-grained version of Bayesian optimization over \( J \)-fold cross validation. We wish to optimize the average performance over all \( J \) folds, but training \( J \) models per experiment is expensive. Instead, we will seek to evaluate only one fold per experiment and infer the performance on the remaining folds using the multi-task model.

With the multi-task GP model, the posterior predictive mean and variance of the average can be written in terms of the predictive mean and variance of each task,

\[
\bar{\mu}(x) = \frac{1}{J} \sum_{j=1}^{J} \mu(x, j),
\]

\[
\bar{v}(x, x') = \frac{1}{J^2} \sum_{j=1}^{J} \sum_{j'=1}^{J} v((x, j), (x', j')).
\]

Since we only evaluate one fold \( j \) for each experiment, we will not have access to \( \bar{y} \); we will need this quantity in order to set the target for EI. To overcome this, we impute observations for any unobserved task \( j' \) and compute an estimate,

\[
\hat{\bar{y}} \approx \frac{1}{J} \left( y_j + \sum_{j' \neq j} \mu(x, j') \right).
\]

Given \( \hat{\bar{y}} \), the challenge now becomes to find both the hyperparameters \( x \) for the next experiment, as well as the fold \( j \) on which to train the model. We choose a \((x, j)\) pair using a two-step heuristic. First, we optimize EI with respect to \( x \) using \( \hat{\bar{y}} \). Then, conditioned on \( x \), we choose the task that yields the highest single-task expected improvement. This can easily be computed using the individual task predictive mean and variance formulas from the multi-task GP.
The problem of minimizing the average error over multiple tasks has been considered in [Bardenet et al., 2013], where they applied Bayesian optimization in order to tune a single model on multiple datasets. Their approach is to project each function to a joint latent space and then iteratively visit each dataset in turn. Another approach can be found in [Hutter et al., 2010], where additional task-specific features are used in conjunction with the inputs $x$ to make predictions about each task.

### 3.5 A Cost-Sensitive Multi-Task Acquisition Function

There are often cases where we might have access to a cheaper auxiliary task that can serve as a proxy for the expensive task that we wish to optimize; we will call this the primary task. For example, the parameters of stochastic gradient descent can first be tuned on a smaller subset of data and then deployed on the full problem [Bottou, 2010]. In these cases, a viable strategy would be to liberally perform experiments on the auxiliary task to gather information, and then only run an expensive experiment on the primary task when we are confident that we will see a good result. In this section, we will devise a strategy to accomplish this in an automated fashion.

Although it is tempting to use the EI criterion, it does not directly generalize to this case. To see this, consider a primary task, labelled 1 and an auxiliary task, labelled 2. A natural strategy is to consider a two-step procedure. First, we evaluate the auxiliary task at a point $x$, and then we condition on this and maximize the expected improvement of the primary task. Let’s call this two-step EI $a_{2EI}((x, 2), (x', 1))$, meaning we condition on observing task 2 at point $x$ and then compute EI on task 1 at $x'$.

$$a_{2EI}((x, 2), (x', 1)) = \int_{f_1} \int_{f_2} P(f_1|\mathcal{D}_N \cup \{(x, 2, f_2)\}, x', 1)P(f_2|\mathcal{D}_N, x, 2) [\max(y_1^* - f_1, 0)] \, df_2 \, df_1$$

$$= \int_{f_1} [\max(y_1^* - f_1, 0)] \int_{f_2} P(f_1|\mathcal{D}_N \cup \{(x, 2, f_2)\}, x', 1)P(f_2|\mathcal{D}_N, x, 2) \, df_2 \, df_1$$

$$= \int_{f_1} P(f_1|\mathcal{D}_N, x', 1) [\max(y_1^* - f_1, 0)] \, df_1$$

$$= a_{EI}(x', 1)$$

Note that without loss of generality, we are ignoring observation noise for convenience. This result follows from the fact that $f_1$ and $f_2$ are jointly Gaussian and therefore when
marginalizing over $f_2$, we can apply the identities in Appendix A.2. Intuitively, this result says that applying EI in this fashion is equivalent to just applying EI without ever observing the auxiliary result.

This is unsatisfactory, however, as clearly the two tasks may be correlated, meaning that observing an auxiliary result will change the posterior distribution over the primary task. We therefore need an acquisition function that is capable of taking advantage of this property.

Luckily, entropy search is one such acquisition function. The reason is that because the tasks are correlated, conditioning on observations of the auxiliary task will change the distribution over the function values of the primary task, hence changing the entropy. In other words, we can reduce uncertainty about the posterior distribution over the primary function and gain knowledge about the location of its minimum by observing values from the auxiliary task.

To make this strategy practical, we assume that all the candidate points come from a fixed set. Following [Hennig and Schuler, 2012], we pick these candidates by taking the top $M$ points according to the EI criterion on the primary task. For each of these candidate points, we can pick the task that maximizes the information gain according to Equation 2.11.

Observe however, that evaluating a point on the auxiliary task can never reveal more information than evaluating the same point on the primary task, meaning that the above strategy would never choose to evaluate a related task. Nevertheless, when cost is taken into account, the auxiliary task may convey more information per unit cost. Thus we translate the entropy search acquisition function to instead reflect the information gain per unit cost of evaluating a candidate point. Once again, let us refer to the primary task using the label 1 and recall that we are maximizing the information gain about the location of the optimum for the primary task. The cost-sensitive entropy search acquisition function for observing task $j$ at point $x$ can be written as,

$$a_{CSES}(x, j) = \frac{H(P_*(x'|D_N, 1)) - \mathbb{E}_{P(y|x', D_N, x, j)}[H(P_*(x'|D_{N \cup \{x, y, j\}}, 1)]}{c(x, j)}.$$ 

where $c(x, j)$, is the positive-valued cost of evaluating task $j$ at $x$. Although, we may not know this cost function in advance, we can estimate it using the multi-task GP to model log $c(x, j)$.

Figure 3.2 shows a visualization of the multi-task information gain per unit cost acquisition function. Here, the objective is to find the minimum of the solid blue function. The green function is an auxiliary objective function. In the bottom of each sub-figure
are lines indicating the expected information gain with regard to the primary objective function. The green dashed line shows the information gain about the primary objective that results from evaluating the auxiliary objective function. Figure 3.2a shows two sampled functions from a GP that are uncorrelated. Evaluating the primary objective produces information gain, but evaluating the auxiliary does not (with the exception of some noise due to Monte Carlo simulation). In Figure 3.2b we see that with two strongly correlated functions, not only do observations on either task reduce uncertainty about the other, but observations from the auxiliary task reveal information about the primary task. Finally, in 3.2c we assume that the primary objective is three times more expensive than the auxiliary task and thus evaluating the related task gives more information gain per unit cost.
Figure 3.2: An example of entropy search with (a): uncorrelated tasks, and (b): correlated tasks. Figure (c) shows the cost-sensitive multi-task entropy search acquisition function.
3.6 Experiments

3.6.1 The Cold Start Problem

Here we compare Bayesian optimization with no initial information to the case where we can leverage results from an already completed optimization on a related task. In each classification experiment the target of Bayesian optimization is the error on a held out validation set. Further details on these experiments can be found in Appendix B.1.

Branin-Hoo The Branin-Hoo function is a common benchmark for optimization techniques [Jones, 2001] that is defined over a bounded set on $\mathbb{R}^2$. As a related task we consider a shifted Branin-Hoo where the function is translated by 10% along either axis. We used Bayesian optimization to find the minimum of the original function and then added the shifted function as an additional task.

Logistic regression We optimize four hyperparameters of logistic regression (LR) on the MNIST dataset [LeCun et al., 1998] using 10000 validation examples. We assume that we have already completed 50 iterations of an optimization of the same model on the related USPS digits task. The USPS data is only $\frac{1}{6}$ the size of MNIST and each image contains $16 \times 16$ pixels, so it is considerably cheaper to evaluate.

Convolutional neural networks on pixels We applied convolutional neural networks\(^1\) (CNNs) to the Street View House Numbers (SVHN) [Netzer et al., 2011] dataset and bootstrapped from a previous run of Bayesian optimization using the same model trained on CIFAR-10 [Krizhevsky, 2009, Snoek et al., 2012]. The SVHN dataset has the same input dimension as CIFAR-10, but is 10 times larger. We used 6000 held-out examples for validation. Additionally, we consider training on $\frac{1}{10}$th of the SVHN dataset to warm-start the full optimization. The best settings yielded $4.77 \pm 0.22\%$ error, which is comparable to domain experts using non-dropout CNNs [Sermanet et al., 2012].

Convolutional networks on k-means features As an extension to the previous CNN experiment, we incorporate a more sophisticated pipeline in order to learn a model for the STL-10 dataset [Coates et al., 2011]. This dataset consists of images with $96 \times 96$ pixels, and each training set has only 1000 images. Overfitting is a significant challenge for this dataset, so we utilize a CNN\(^2\) on top of k-means features in a similar approach.

---

\(^1\)Using the Cuda Convnet package: https://code.google.com/p/cuda-convnet

\(^2\)Using the Deepnet package: https://github.com/nitishsrivastava/deepnet
to [Gens and Domingos, 2012], as well as dropout [Hinton et al., 2012b]. We bootstrapped Bayesian optimization using the same model trained on CIFAR-10, which had achieved 14.2% test error on that dataset. During the optimization, we used the first fold for training, and the remaining 4000 points from the other folds for validation. We then trained separate networks on each fold using the best hyperparameter settings found by Bayesian optimization. Following reporting conventions for this dataset, the model achieved 70.1 ± 0.6% test-set accuracy.

The results of these experiments are shown in Figure 3.3a-b. In each case, the multi-task optimization finds a better function value much more quickly than single-task optimization. Clearly there is information in the related tasks that can be exploited. To better understand the behaviour of the different methods, we plot the average cumulative error (ACE), i.e., the average of all function values seen up to a given time, in Figures 3.3e and 3.3f. The single-task method wastes many more evaluations exploring poor hyperparameter settings. In the multi-task case, this exploration has already been performed and more evaluations are spent on exploitation.

As a baseline (the dashed black line), we took the best model from the first task and applied it directly to the task of interest. For example, in the CNN experiments this involved taking the best settings from CIFAR-10. This “direct transfer” performed well in some cases and poorly in others. In general, we have found that the best settings for one task are usually not optimal for the other.

### 3.6.2 Fast Cross-Validation

$J$-fold cross-validation is a widely used technique for estimating the generalization error of machine learning models, but requires retraining a model $J$ times. This can be prohibitively expensive with complex models and large datasets. It is reasonable to expect, however, that if the data are randomly partitioned among folds that the errors for each fold will be highly correlated. For a given set of hyperparameters, we can therefore expect diminishing returns in estimating the average error for each subsequently evaluated fold. With a good GP model, we can very likely obtain a high quality estimate by evaluating just one fold per setting. In this experiment, we apply the algorithm described in Section 3.4 in order to dynamically determine which points/folds to query.

We demonstrate this procedure on the task of training probabilistic matrix factorization (PMF) models for recommender systems [Salakhutdinov and Mnih, 2008]. The hyperparameters of the PMF model are the learning rate, an $\ell_2$ regularizer, the number of latent factors (matrix rank), and the number of epochs. We use 5-fold cross validation...
on the Movielens-100k dataset [Herlocker et al., 1999]. In Figure 3.4a we show the best error obtained after a given number of function evaluations as measured by the number of folds queried, averaged over 50 optimization runs. For the multi-task version, we show both the true average cross-validation error, as well as the estimated error according to the GP. In the beginning, the GP fit is highly uncertain, so the optimization exhibits some noise. As the GP model becomes more certain however, the true error and the GP estimate converge and the search proceeds rapidly compared to the single-task counterpart. In Figure 3.4b, we show the best observed error after a given number of function evaluations on a randomly selected run. For a particular fold, the error cannot improve unless that fold is directly queried. The algorithm makes nontrivial decisions in terms of which fold to query, steadily reducing the average error.

### 3.6.3 Using Small Datasets to Bootstrap Training on Large Datasets

As a final empirical analysis, we evaluate the dynamic multi-task entropy search strategy developed in Section 3.5 on two hyperparameter tuning problems. We treat the cost, $c(x, j)$, of a function evaluation as being the real running time of training and evaluating the machine learning algorithm with hyperparameter settings $x$ on task $j$. We assume no prior knowledge about either task, their correlation, or their respective cost, but instead estimate these as the optimization progresses. In both tasks we compare using our multi-task entropy search strategy (MTBO) to optimizing the task of interest independently (STBO).

First, we revisit the logistic regression problem from Section 3.6.1 (Figure 3.3b) using the same experimental protocol, but rather than assuming that there is a completed optimization of the USPS data, the Bayesian optimization routine can instead dynamically query USPS as needed. Figure 3.5e, shows the average time taken by either strategy to reach the values along the blue line. We see that MTBO reaches the minimum value on the validation set within 40 minutes, while STBO reaches it in 100 minutes. Figures 3.5a and 3.5b show that MTBO reaches better values significantly faster by spending more function evaluations on the related, but relatively cheaper task.

Next, we evaluate the very expensive problem of optimizing the hyperparameters of online Latent Dirichlet Allocation [Hoffman et al., 2010] on a large corpus of 200,000 documents. [Snoek et al., 2012] demonstrated that on this problem, Bayesian optimization could find better hyperparameters in significantly less time than the grid search conducted by the authors. We repeat this experiment here using the exact same grid as
[Snoek et al., 2012] and [Hoffman et al., 2010] but provide an auxiliary task involving a subset of 50,000 documents and 25 topics on the same grid. Each function evaluation on the large corpus took an average of 5.8 hours to evaluate while the smaller corpus took 2.5 hours. We performed our multi-task Bayesian optimization restricted to the same grid and compare to the results of the standard Bayesian optimization of [Snoek et al., 2012] (the GP EI MCMC algorithm). In Figure 3.5f, we see that our MTBO strategy finds the minimum in approximately 6 days of computation while the STBO strategy takes 10 days. Our algorithm saves almost 4 days of computation by being able to dynamically explore the cheaper alternative task. We see in Figure 3.5d that particularly early in the optimization, the algorithm explores the cheaper task to gather information about the expensive one.

3.7 Discussion and Conclusion

In this chapter we have introduced an extention of Bayesian optimization that allows one to transfer information from related problems to a problem of interest. Multi-task Bayesian optimization utilizes multi-task Gaussian processes in order to learn a covariance that determines the relationship between different problems.

We explored three different applications of this extension: solving the cold-start problem, where information from old problems can be used to inform new problems, speeding up cross-validation by avoiding redundant multi-fold evaluations, and using a cheap problem to inform an expensive problem in order to reduce the total optimization time.

Beyond these applications, there are a number of interesting connections to recent work as well as several avenues that merit further exploration.

First, it has been observed that the techniques developed here are useful in the constrained setting [Gelbart et al., 2014] when the constraints are decoupled. This is where the objective and constraint can be evaluated separately and therefore it can be beneficial to treat them as individual tasks. In [Hernández-Lobato et al., 2015], predictive entropy search criterion [Hernández-Lobato et al., 2014] was used instead of entropy search and found to perform better. This acquisition function could be applied to the problems outlined here and may provide a further boost in performance.

Practically speaking, it may be possible that each task has its own observation noise and prior mean, in which case it would be beneficial to learn these separately. The structure of the task covariance that we use in our experiments already implicitly assigns a different kernel amplitude to each task. We give an example of learning task-specific kernel parameters in Chapter 4.
Another practical trick that we did not explore is the re-use of the best hyperparameters from old tasks in order to initialize new tasks, rather than using a random initial design. Indeed, [Feurer et al., 2015] showed that this can be a highly effective strategy.

In our experiments, we assumed that there we had no information about the relationship between tasks other than that we believed they were similar. It is certainly possible to include extra covariates within the multi-task kernel, such as data set size. Examples of this can be found in contextual multi-task optimization methods developed in [Hutter et al., 2010] and [Krause and Ong, 2011]. The particular example of dataset size is reminiscent of techniques found in [Feurer et al., 2015].

Perhaps the biggest drawback of multi-task Bayesian optimization is the $O(N^3)$ scaling of the Gaussian process. In our applications and experiments, we showed that the multi-task framework can be effective with a small number of tasks, however when the number of tasks grows then it can be expected that the total number of observations will grow as well. One way to overcome the scaling issue is to use scalable techniques such as sparse Gaussian process kernels [Snelson and Ghahramani, 2005; Nickson et al., 2014] or deep neural networks [Snoek et al., 2015].
Figure 3.3: (a)-(b): validation error per function evaluation. (e), (f): average cumulative error (ACE) over function evaluations. The dashed black line is the best settings from the initial task applied to the new task.
Figure 3.4: (a): PMF cross-validation error per function evaluation on Movielens-100k. (b): lowest error observed for each fold per function evaluation for a single run.
Figure 3.5: (a), (c): validation error as a function of time spent training the models. (b), (d): validation error over the number of function evaluations. (e), (f): time taken to reach a given validation error.
Chapter 4

Input Warping

4.1 Introduction

When tuning the hyperparameters of a model, researchers often first transform the input space using a monotonic function such as the natural logarithm and then perform a grid search in this transformed space. Such an optimization in “log-space” takes advantage of a priori knowledge that the model is sensitive to small changes in some regions, and less sensitive in others. For example, when optimizing the hyperparameters of a machine learning algorithm, we might expect the objective function to be more sensitive near the optimum, but less sensitive far away from the optimum. That is, we would expect bad hyperparameters to yield similarly bad performance everywhere (e.g., classifying at random) but expect the generalization performance to be sensitive to small tweaks in good hyperparameter regimes. As another example, when adjusting the learning rate, moving from 0.01 to 0.02 represents a 100% increase, while moving from 0.1 to 0.11 represents only a 10% increase. We would therefore expect the model to be far more sensitive to absolute changes in the smaller regime than the larger. This phenomenon is known as non-stationarity, and the ability to determine where the model is sensitive to hyperparameters is critical to proper tuning. In this chapter, our goal is to replicate this ability to detect and compensate for these changes in sensitivity within the Bayesian optimization framework.

A major limitation of the most commonly used Gaussian process kernels is the assumption of stationarity—that the covariance between two outputs is invariant to translations in input space. Or more precisely, that the kernel can be written as a function of the absolute difference between its inputs, $|x - x'|$. This assumption simplifies the regression task, but hurts the ability of the Gaussian process to model more realistic non-stationary processes. This presents a challenge for Bayesian optimization, as many
Chapter 4. Input Warping

problems of interest are inherently non-stationary.

We introduce a simple solution that allows Gaussian processes to model a large variety of non-stationary processes and is particularly well suited to Bayesian optimization. We automatically learn a bijective warping of the inputs that removes major non-stationary effects. This is achieved by projecting each dimension of the input through the cumulative distribution function of the beta distribution, while marginalizing over the shape of the warping. Our approach is computationally efficient, captures a variety of desirable transformations, and is easily interpretable. In the context of Bayesian optimization, understanding the parameter space is often just as important as achieving the best possible result and our approach lends itself to a straightforward analysis of the non-stationarities in a given problem domain. We further extend this idea to multi-task Bayesian optimization so that multiple tasks can be warped into a jointly stationary space in order to better take advantage of their shared structure.

In our experiments, we show that modeling non-stationarity is extremely important and yields significant empirical improvements in the performance of Bayesian optimization. For example, we show that on a recently introduced optimization benchmark [Eggensperger et al., 2013], our method outperforms all of the previous state-of-the-art algorithms on the problems with continuous-valued parameters. We further observe that on four different challenging machine learning optimization tasks our method outperforms that of [Snoek et al., 2012], consistently converging to a better result in fewer function evaluations. As our methodology involves a transformation of the inputs, this strategy generalizes to a wide variety of models and algorithms. Empirically, modeling non-stationarity is a fundamentally important component of effective Bayesian optimization.

4.2 Non-stationary Gaussian Process Regression

4.2.1 Input Warping

The idea behind input warping is to change the kernel function to be \( k(w(x), w(x')) \), where \( w: \mathcal{X} \to [0, 1]^D \) is a bijective warping function. We choose an element-wise warping function that applies a separate warping to each dimension \( d \) of the input. In particular, we consider the following warping function,

\[
w_d(x_d) = \int_0^{x_d} \frac{u^{\alpha_d-1}(1-u)^{\beta_d-1}}{B(\alpha_d, \beta_d)} \, du,
\]
**Chapter 4. Input Warping**

Exponential decay

A non-stationary periodic function

Two examples of how input warping using the beta CDF can transform a non-stationary function into a stationary one. The warping function maps the original inputs on the horizontal axis to new inputs shown on the vertical axis. The effect is to stretch and contract regions of the input space in such a manner as to remove non-stationarity.

where \( B(\alpha_d, \beta_d) \) is the beta function. That is, \( w_d(x_d) \) is the cumulative distribution function of the beta distribution with distinct shape parameters \( \alpha_d > 0 \) and \( \beta_d > 0 \). Alternatively, one can think of input warping as applying a particular kind of non-stationary kernel to the original data. Examples of difficult functions and ideal corresponding beta warpings are shown in Figure 4.1.

Our choice of the CDF of the beta distribution is motivated by the fact that it is capable of expressing a variety of monotonic warpings, while still being concisely parameterized. In general, there are many other suitable choices. For example, the Kumaraswamy CDF,

\[
  w_d(x_d) = 1 - (1 - x_d^{\alpha_d})^{\beta_d}.
\]

The Kumaraswamy distribution is often used as a substitute for the beta distribution for computational reasons, however this cost is negligible in the settings we consider, so we will use the beta CDF.

### 4.2.2 Integrating over Warpings

Rather than assume a single, explicit transformation function, we define a hierarchical Bayesian model by placing a prior over the shape parameters, \( \alpha \) and \( \beta \) of the beta distribution and integrating them out using slice sampling, following the treatment of
covariance hyperparameters in [Snoek et al., 2012]. We use a log-normal distribution, i.e.,

\[
\log(\alpha) \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2), \\
\log(\beta) \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2),
\]

(4.1) (4.2)

to express a prior for a wide family of desirable functions. Figure 4.2 demonstrates example warping functions arising from sampling beta parameters from various instantiations of the prior. Note that the geometric mean, or median of the zero-mean log-normal distribution for \(\alpha\) and \(\beta\) corresponds to the identity transform. With this prior the model assumes no transformation of the input space without evidence to the contrary. In the following experiments we use this formulation with a variance of 0.75, assuming no prior knowledge of the form of the transformation for a particular problem. However, a nice property of this formulation is that a user can easily specify a prior for a specific form of warping, as we show in Figure 4.2.
Figure 4.2: Each figure shows 50 warping functions resulting from the beta CDF where the shape parameters $\alpha$ and $\beta$ are sampled from a log-normal prior with a different mean and variance. The flexible beta CDF captures many desirable warping functions and adjusting the prior over input warpings allows one to easily encode prior beliefs over the form of non-stationarity. The parameters of the prior distribution, expressed as $(\mu_\alpha, \sigma^2_\alpha, \mu_\beta, \sigma^2_\beta)$ are, (a): $(0.0, 0.75, 0.0, 0.75)$, (b): $(1.5, 0.75, 0.0, 0.75)$, (c): $(0.0, 0.75, 1.5, 0.75)$, (d): $(1.5, 0.75, 1.5, 0.75)$, (e): $(-1.0, 0.75, -1.0, 0.75)$. 


4.2.3 Multi-Task Input Warping

When training the same model on different datasets, certain properties, such as the size of the dataset, can have a dramatic effect on the optimal hyperparameter settings. For example, a model trained on a small dataset will likely require more regularization than the same model trained on a larger dataset. In other words, it is possible that one part of the input space on one task can be correlated with a different part of the input space on another task. To account for this, we allow each task to have its own set of warping parameters. Inferring these parameters will effectively try to warp both tasks into a jointly stationary space that is more suitably modelled by a standard multi-task kernel. In this way, large values on one task can map to small values on another, and vice versa.

4.2.4 Other Methods for Non-Stationary Gaussian Process Regression

Numerous approaches have been proposed to extend GPs to model non-stationary functions. [Gramacy, 2005] proposed a Bayesian treed GP model which accommodates various complex non-stationarities through modeling the data using multiple GPs with different covariances. Various non-stationary covariance functions have been proposed, e.g., [Higdon et al., 1998, Rasmussen and Williams, 2006]. Previously, [Sampson and Guttorp, 1992] proposed projecting the inputs into a stationary latent space using a combination of metric multidimensional scaling and thin plate splines. [Schmidt and O’Hagan, 2003] extended this warping approach for general GP regression problems using a flexible GP mapping. Spatial deformations of two dimensional inputs have been studied extensively in the spatial statistics literature [Anderes and Stein, 2008]. [Bornn et al., 2012] project the inputs into a higher dimensional stationary latent representation. [Snelson et al., 2004] apply a warping to the output space, while [Adams and Stegle, 2008] perform input-dependent output scaling with a second Gaussian process.

Compared to these approaches, our approach is relatively simple, yet as we will demonstrate, flexible enough to capture a wide variety of non-stationary behaviours. Our principal aim is to show that addressing non-stationarity is a critical component of effective Bayesian optimization, and that any advantages gained from using our approach would likely generalize to more elaborate techniques.
4.3 Experiments

Our empirical analysis is comprised of three distinct experiments. In the first experiment, we compare to the method of [Snoek et al., 2012] in order to demonstrate the effectiveness of input warping. In the second experiment, we compare to other hyperparameter optimization methods using a subset of the benchmark suite found in [Eggensperger et al., 2013]. Finally, we show how the multi-task extension can provide further improvements.

4.3.1 Comparison to Stationary GPs

Experimental setup We evaluate the standard Gaussian process expected improvement algorithm (GP EI MCMC) as implemented by [Snoek et al., 2012], with and without warping. Following their treatment, we use the Matérn-5/2 kernel and we marginalize over kernel parameters $\Theta$ using slice sampling [Murray and Adams, 2010]. We repeat three of the experiments\(^1\) from [Snoek et al., 2012], and perform an experiment involving the tuning of a deep convolutional neural network\(^2\) on a subset of the popular CIFAR-10 data set [Krizhevsky, 2009]. The deep network consists of three convolutional layers and two fully connected layers and we optimize over two learning rates, one for each layer type, six dropout regularization rates, six weight norm constraints, the number of hidden units per layer, a convolutional kernel size and a pooling size for a total of 21 hyperparameters. On the logistic regression problem we also compare to warping the input space a priori using the log-transform (optimizing in log-space).

Results Figure 4.3 shows that in all cases, dealing with non-stationary effects via input warpings significantly improves the convergence of the optimization. In particular, we notice on the higher-dimensional convolutional network problem a profound improvement (Figure 4.3d) when the non-stationarities are inferred.

In Figure 4.4 we plot examples of some of the inferred warpings. For logistic regression, Figure 4.4a shows that our method learns different logarithmic-like warpings for three dimensions and no warping for the fourth. Figure 4.4b shows how the posterior distribution over the learning rate warping evolves, becoming more extreme and more certain, as observations are gathered. Figure 4.4c shows that on both convolutional and dense layers, the intuition that one should log-transform the learning rates holds. For transformations on weight norm constraints, shown in Figure 4.4d, the weights connected to the inputs and outputs use a sigmoidal transformation, the convolutional-layer weights

\(^1\)See [Snoek et al., 2012] for details of these experiments.

\(^2\)We use the Deepnet package from https://github.com/nitishsrivastava/deepnet
Figure 4.3: An empirical comparison of Bayesian optimization following the standard Gaussian process expected improvement algorithm (GP EI MCMC) and our strategy (Warped GP EI MCMC) for modeling input non-stationarity. The methods are compared on four challenging problems involving the optimization of the hyperparameters of popular machine learning algorithms.
use a convex transformation, and the dense-layer weights use a concave transformation. Effectively, this means that the most variation in the error occurs in the medium, high and low scales respectively for these types of weights. Especially interesting are the wide variety of transformations that are learned for dropout on different layers, shown in Figure 4.4e. These show that different layers benefit from different dropout rates, which was also confirmed on test set error, and suggests that they should not necessarily be set to 0.5 [Hinton et al., 2012b].

It is clear that the learned warpings are non-trivial. In some cases, like with learning rates, they agree with intuition, while for others like dropout they yield surprising results. Given the number of hyperparameters and the variety of transformations, it is highly unlikely that even experts would be able to determine the whole set of appropriate warpings. This highlights the utility of learning them automatically.

### 4.3.2 HPOLib Continuous Benchmarks

In our next set of experiments, we tested our approach on the subset of benchmarks over continuous inputs from the HPOLib benchmark suite [Eggensperger et al., 2013].
These benchmarks are designed to assess the strengths and weaknesses of several popular hyperparameter optimization schemes. All of the tested methods perform Bayesian optimization, however the underlying surrogate models differ significantly. The SMAC package [Hutter et al., 2010] uses a random forest, the Hyperopt package [Bergstra et al., 2011] uses the tree Parzen estimator, and the Spearmint package [Snoek et al., 2012] uses a Gaussian process. For our experiments, we augmented the Spearmint package with input warping.

**Results**  We evaluate our algorithm on the continuous-valued parameter benchmarks proposed in [Eggensperger et al., 2013] and the results are shown in Table 4.1. We compare to Sequential Model Based Algorithm Configuration (SMAC) [Hutter et al., 2010], the Tree Parzen Estimator (TPE) [Bergstra et al., 2011] and Bayesian optimization with GPs (BO) [Snoek et al., 2012]. The results for SMAC, BO, and TPE are taken from [Eggensperger et al., 2013]. Following the standard protocol for these benchmarks, each algorithm was run ten times for the given number of evaluations, and the average validation loss and standard deviation are reported. The algorithm with the lowest validation loss is shown in bold. We note that on some of the benchmarks, warped Bayesian optimization converges to a solution in far fewer evaluations than the protocol allows. The number of evaluations is shown in Table 4.2

Overall, input warpings improve the performance of the Gaussian process approach such that it does at least as well as every other method, and in many cases better. Furthermore, the standard deviation also decreases significantly in many instances, meaning that the results are far more reliable.

Interestingly, the random forest approach in SMAC also naturally deals with non-stationarity, albeit in a fundamentally different way, by partitioning the space in a non-uniform manner. There are several possibilities to explain the performance discrepancy. Unlike random forests, Gaussian processes produce a smooth function of the inputs, meaning that EI can be locally optimized via gradient methods, so it is possible that better query points are selected in this way. Alternatively, the random forest is not a well-defined prior on functions and there may be overfitting in the absence of parameter marginalization. It may simply be the case that the form of non-stationarity captured by random forests does not reflect the non-stationarity found in these problems. Further investigation is required to tease apart this discrepancy.
Table 4.1: A comparison of hyperparameter optimization methods on the continuous benchmark problems from [Eggensperger et al., 2013].

<table>
<thead>
<tr>
<th>Experiment</th>
<th>SMAC</th>
<th>BO</th>
<th>TPE</th>
<th>Warped BO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branin (0.398)</td>
<td>0.655 ± 0.27</td>
<td><strong>0.398 ± 0.00</strong></td>
<td>0.526 ± 0.13</td>
<td><strong>0.398 ± 0.00</strong></td>
</tr>
<tr>
<td>Hartmann 6 (-3.322)</td>
<td>−2.977 ± 0.11</td>
<td>−3.133 ± 0.41</td>
<td>−2.823 ± 0.18</td>
<td><strong>−3.3166 ± 0.02</strong></td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>8.6 ± 0.9</td>
<td>7.3 ± 0.2</td>
<td>8.2 ± 0.6</td>
<td><strong>6.88 ± 0.0</strong></td>
</tr>
<tr>
<td>LDA (On grid)</td>
<td>1269.6 ± 2.9</td>
<td>1272.6 ± 10.3</td>
<td>1271.5 ± 3.5</td>
<td><strong>1266.2 ± 0.1</strong></td>
</tr>
<tr>
<td>SVM (On grid)</td>
<td><strong>24.1 ± 0.1</strong></td>
<td>24.6 ± 0.9</td>
<td>24.2 ± 0.0</td>
<td><strong>24.1 ± 0.1</strong></td>
</tr>
</tbody>
</table>

Table 4.2: Number of evaluations required for warping to converge to the optimal solution.

<table>
<thead>
<tr>
<th>Experiment</th>
<th># Evals</th>
<th># Warping Evals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branin</td>
<td>200</td>
<td>40</td>
</tr>
<tr>
<td>Hartmann 6</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>LDA (On grid)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>SVM (On grid)</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

### 4.3.3 Multi-Task Warping Experiments

In this experiment, we apply multi-task warping to the logistic regression and online LDA problems from Chapter 3. In the logistic regression problem, a search over hyperparameters has already been completed on the USPS dataset, which consists of 6,000 training examples of handwritten digits of size $16 \times 16$. It was demonstrated in Chapter 3 that it is possible to use this previous search to speed up the hyperparameter search for logistic regression on the MNIST dataset, which consists of 60,000 training examples of size $28 \times 28$.

In the online LDA problem, we assume that the model has been trained on 50,000 documents and that we would now like to train it on 200,000 documents. Again, it was shown that it is possible to transfer information over to this task, resulting in more efficient optimization.

**Results**  In Figure 4.5c, we see that warped multi-task Bayesian optimization (warped MTBO) reaches the optimal solution faster than the non-warped version (MTBO), and both multi-task methods outperform single-task Bayesian optimization (STBO). On the logistic regression problem shown in Figure 4.5b, it appears that ordinary MTBO actually gets stuck in a local minimum, while warped MTBO is able to consistently escape this
Chapter 4. Input Warping

Figure 4.5: Multi-task warping applied to logistic regression and online LDA. For logistic regression, the model is trained on USPS first and then the search is transferred to MNIST. For online LDA, the data is subsampled and the model is learned, then the search transferred to the full dataset. In Figure 4.5a, we show the mean warping learned by each task for each parameter. The solid lines indicates the MNIST task, while the dashed lines indicate the USPS task.

by the 20\textsuperscript{th} function evaluation.

In Figure 4.5a we show the mean warping learned for each task/hyperparameter combination (generated by averaging over samples from the posterior). The model trained on USPS is more sensitive to e.g., the L\textsubscript{2} regularizer setting than with MNIST, as evidenced by the more extreme warping. This intuitively makes sense, as the larger MNIST dataset is typically less prone to overfitting. Also interesting, but less intuitive is that logistic regression on USPS is relatively insensitive to the number of learning epochs, but very sensitive to the batch size, while on MNIST it is relatively insensitive to the batch size, but very sensitive to the learning epochs.
4.4 Discussion

4.4.1 The Boundary Issue

In moderate to high dimensions, Bayesian optimization can spend an inordinate amount of time exploring the boundaries of the search space. The reason for this is a combination of the way uncertainty is represented by the GP and the curse of dimensionality. For a practitioner this is less than ideal. The reason is that when a practitioner sets the boundaries of the search space, often they have put some thought into setting those boundaries appropriately. Usually, the intention is to ensure that the right values lie somewhere near the center, which is a very explicit prior assumption. Here, we will discuss two ways in which we can alleviate the boundary issue.

In terms of how a GP represents uncertainty, notice that the predictive variance in Equation 2.8 is maximized when the query point is as far from the training data as possible. This means that given any point in the unit hypercube, the farthest point from it will be on a corner of the hypercube. Therefore, if the mean is not particularly informative so that EI is being driven by the variance, then the search will likely place observations on the edges of the hypercube. Once enough points have been observed, then the points with the highest variance might lie at the interior. This effect is exacerbated in higher dimensions as there will be exponentially many more corner points.

Indeed, even random search, a common hyperparameter search technique, will suffer the boundary issue due to the curse of dimensionality. Consider drawing a point uniformly at random in a $D$ dimensional space, then the probability that none of the dimensions lies in within a small distance $\epsilon$ of the boundary will be equal to $(1 - 2\epsilon)^D$. For example, in a 20-dimensional space, the probability that no dimensions lie within 0.05 units of the boundary is a little more than 10%. Meaning roughly nine out of every ten suggestions will lie near a boundary.

We can solve this in two complimentary ways. The first is to put a sigmoidal prior on the warping parameters by setting $\mu_\alpha > 1$ and $\mu_\beta > 1$ in Equations 4.1 and 4.2. This contracts the hypercube around the boundaries, reducing the variance in these areas. This allows the search to place enough initial points on the interior of the hypercube that subsequent evaluations are far less likely to be on the boundary, even if the prior is incorrect.

We can solve this in two complimentary ways. The first is to put a sigmoidal prior on the warping parameters by setting $\mu_\alpha > 1$ and $\mu_\beta > 1$ in Equations 4.1 and 4.2. This contracts the hypercube around the boundaries, reducing the variance in these areas. This allows the search to place enough initial points on the interior of the hypercube that subsequent evaluations are far less likely to be on the boundary, even if the prior is incorrect.

The second approach is to place quadratic mean prior over the space [Snoek et al., 2015]. That is, we can use the following axis-aligned quadratic mean function:

$$m(x) = a + (x - c)^T B(x - c),$$
where $a$ is a constant offset, $c$ is the center of the quadratic, and $B$ is a positive diagonal scaling matrix. We recommend placing a Horseshoe prior [Carvalho et al., 2009] on the scaling factors. This is a spike and slab prior, so that if the quadratic is misspecified along some dimension then the posterior scaling factor can be brought close to 0 and the quadratic can be ignored along that dimension.

### 4.4.2 Future Directions

There are a number of interesting directions in which to expand on this research. A natural avenue is to explore more elaborate warping functions. The beta distribution is a bimodal distribution, but it would certainly be possible to use warpings based on multi-modal distributions. If we relax the assumptions that the warping must be axis-aligned and bijective, then this opens up even more possibilities. An example of a joint, non-bijective warping can be found in [Snoek et al., 2015], where we apply a deep neural network.

Although we focused on non-stationarity with respect to the inputs, there are other ways in which these effects can manifest. For example, there has been work on modelling warping with respect to the output [Snelson et al., 2004], which can help deal with discontinuities in the function. There has also been work on modelling non-stationarity with respect to the amplitude of the function, where the magnitude of the function changes over the input domain [Adams and Stegle, 2008]. Finally, a form of non-stationarity that appears frequently in real-world problems is in the noise process. This effect is known as heteroscedasticity. There is a wide literature on dealing with heteroscedasticity for GPs (e.g., [Le et al., 2005] and references therein), and there have been some preliminary attempts at dealing with it in Bayesian optimization [Assael et al., 2014].

### 4.5 Conclusion

In this chapter we addressed the issue of non-stationary effects in hyperparameter optimization. Essentially, in typical machine learning models, small changes to a hyperparameter can have disproportionate effects on performance, depending on the value of the hyperparameter. Experts deal with this issue by applying clever transformations to the hyperparameter space, and they choose these transformations by intuition and heuristics.

Our aim was to incorporate this idea into the Bayesian optimization framework. Traditional GP-based optimization does not naturally account for non-stationary effects because GPs tend to rely on stationary kernels. To remedy this, we proposed to use
input warping before applying the kernel function. In particular, taking advantage of the fact that the search is carried out within the unit hypercube, we used the CDF of the beta distribution due to its ability to model a wide range of desirable transformations. We used MCMC inference to integrate out the warping parameters in order to account for uncertainty when the number of observations is small. This also gives the user the ability to express priors on the warping. As an alternative that we did not explore here, a user can simply transform their variables in advance, and rely on the warping to adjust this as needed.

We demonstrated empirically that warping almost always benefits Bayesian optimization on practical problems. We further showed that in the multi-task case, applying a separate warping to each task can correct the issue that the same model might be sensitive to different regions on different problems. Another benefit of the warping approach is it allows us to determine where the objective is sensitive to hyperparameter settings. This kind of analysis can perhaps help to inform more robust models, as a solution in a stable region would likely be preferable to a similarly performing solution in an unstable region.
Chapter 5

Architecture Search

5.1 Introduction

So far we have considered tuning hyperparameters that fall into three general types: floating-point, integers, and categorical variables. In this chapter we will consider a new type called a conditional hyperparameter. A conditional hyperparameter is a boolean or integer-valued variable whose value determines the relevance of other hyperparameters. A hyperparameter is relevant if its setting affects the performance of the model.

As a concrete example, consider the number of layers in an MLP. If each layer is given its own set of hyperparameters such as the number of units, learning rate, regularization strength, etc., then the depth variable determines whether these hyperparameters will be relevant. If the depth is set to 1, then any hyperparameter corresponding to the second layer and above will be irrelevant, since those layers will not be used in the network. In this case, we say that the depth parameter helps to determine the architecture of the network. The goal then is to simultaneously tune the architecture, along with the other hyperparameters.

Conditional hyperparameters can be found in many other areas of machine learning. When building ensembles, conditional variables can be used to determine which models get incorporated into the ensemble. Conditional variables can also be used to select amongst various data pre-processing methods.

There are approaches that have been built to handle the conditional problem. One approach is random search [Bergstra and Bengio, 2012], which deals with the problem trivially since we can just ignore irrelevant variables in its proposals. The tree parzen estimator is also designed to deal with this problem by modelling the conditional space using a conditional generative model. Finally, Bayesian optimization using random forests can ignore irrelevant variables in the random forest decision process. Here, we would
like to incorporate the ability to deal with conditional spaces into GPs in order to take advantage of their desirable properties.

Conditional variables are difficult for GP-based Bayesian optimization because they represent variable-length inputs, assuming irrelevant variables are not incorporated into the input. Standard kernels operate on a Euclidean distance metric and this is not defined on vectors of variable sizes.

In spite of this, it is reasonable to expect that the same hyperparameters will behave similarly across different architectures. This is similar to the multi-task intuition, where different architectures can be thought of as different tasks. We expect that on shared relevant hyperparameters, the optimal hyperparameter settings will also be similar, or that bad settings will be bad across architectures. The key then is to define a kernel that can operate on conditional spaces.

In the rest of this chapter, we outline one such kernel. We start by defining some intuitive properties that the kernel should possess, and proceed to develop a kernel that satisfies these properties. The benefit of this particular approach for Bayesian optimization remains inconclusive, however the insights gained here pave the way for future approaches.

### 5.2 Binarizing Conditional Variables

Although we defined conditional variables as boolean or integer-valued, we can consider only the case of boolean variables without loss of generality by converting an integer variable into a sequence of boolean variables where an integer value of \(i\) implies that all boolean variables prior to \(i\) are set to True and all boolean variables after are set to False. This concept is illustrated in Figure 5.1 for the case of the depth parameter in an MLP.

### 5.3 Simple Solutions

Before we delve into a more principled approach to the architecture search problem, we outline some simple and practical ways in which current Bayesian optimization systems can deal with this issue.

The first, and simplest approach is to treat each architecture as independent and apply an individual GP to each one. This approach was used in [Bergstra et al., 2011]. The main issue with this approach is that it does not share information across architectures. It also requires a different model for each architecture, which becomes infeasible if the total number of architectures grows exponentially with the number of conditional variables.
Another simple approach is to set irrelevant variables to some default value. An example of this approach for multi-task neural networks is used in [Dahl et al., 2014]. This allows all architectures to be expressed in the same vector space, but imposes very strong assumptions on the parameter space that relevant variables are most similar to their irrelevant counterparts if they are close to the default value. We experiment with this method, along with an alternative solution that selects random values for irrelevant variables.

Lastly, it is possible to tie the hyperparameters across architectures, e.g., tying the hyperparameters across layers. This reduces the size of the search space, but is not applicable to all conditional spaces and reduces the flexibility of the model.

5.4 A Kernel for Conditional Parameter Spaces

GPs employ a positive-definite kernel function to model the covariance between function values. Typical GP models cannot, however, model the covariance between function values whose inputs have different (possibly overlapping) sets of relevant variables.

In this section, we construct a kernel between points in a space that may have dimensions which are irrelevant under known conditions (further details are available in [Hutter and Osborne, 2013]).

We define functions $\delta_i: \mathcal{X} \rightarrow \{\text{true, false}\}$, for $i \in \{1, \ldots, D\}$. $\delta_i(x)$ stipulates the relevance of the $i$th feature $x_i$ to the function value $f(x)$. 
5.4.1 Intuitive Properties of a Conditional Embedding

Imagine trying to model the performance of a neural network having either one or two hidden layers, with respect to the weight decay parameter for each layer, $\lambda_1$ and $\lambda_2$. For the purposes of maintaining the notation from earlier, we will refer to these as $x_1$ and $x_2$. If $y$ represents the performance of a one layer-net, then the value $x_2$ doesn’t matter, since there is no second layer to the network. Below, we’ll write an input triple as $(x_1, \delta_2(x), x_2)$ and assume that $\delta_1(x) = \text{true}$; that is, the regularization parameter for the first layer is always relevant.

In this setting, we want the kernel $k$ to be dependent on which hyperparameters are relevant, and the values of relevant hyperparameters. For parameters that are irrelevant, we would like to avoid any bias that the aforementioned default value approach would incur. Consider two different settings of the first-layer hyperparameters $x_1$ and $x'_1$, then we can formally state this intuition in two basic properties that our kernel should satisfy:

- If we are comparing two points for which the same hyperparameters are relevant, the value of any unused (irrelevant) hyperparameters shouldn’t matter,

  $$k((x_1, \text{false}, x_2), (x'_1, \text{false}, x'_2)) = k((x_1, \text{false}, x''_2), (x'_1, \text{false}, x''''_2)), \forall x_2, x'_2, x''_2, x''''_2.$$  

- The covariance between a point with two relevant hyperparameters and a point with only one should again only depend on their shared relevant hyperparameters,

  $$k((x_1, \text{false}, x_2), (x'_1, \text{true}, x'_2)) = k((x_1, \text{false}, x''_2), (x'_1, \text{true}, x''''_2)), \forall x_2, x'_2, x''_2, x''''_2.$$  

Put another way, in the absence of any other information, these properties encode our prior ignorance about the irrelevant (missing) parameters while still allowing us to model correlations between relevant parameters.

5.4.2 Cylindrical Embedding

We can build a kernel with these properties for each possibly irrelevant input dimension $i$ by embedding our points into a higher-dimensional Euclidean space. Specifically, the embedding we use is:

$$g_i(x) = \begin{cases} [0, 0]^T & \text{if } \delta_i(x) = \text{false} \\ \omega_i[\sin \pi \rho_i x_i, \cos \pi \rho_i x_i]^T & \text{otherwise.} \end{cases} \quad (5.1)$$

Where $\omega_i \in \mathbb{R}^+$ and $\rho_i \in [0, 1]$. 
Figure 5.2: A demonstration of the embedding giving rise to the pseudo-metric. All points for which \( \delta_2(x) = \text{false} \) are mapped onto a line varying only along \( x_1 \). Points for which \( \delta_2(x) = \text{true} \) are mapped to the surface of a semicylinder, depending on both \( x_1 \) and \( x_2 \). This embedding gives a constant distance between pairs of points which have differing values of \( \delta \) but the same values of \( x_1 \).

Figure 5.2 shows a visualization of the embedding of points \((x_1, \delta_2(x), x_2)\) into \( \mathbb{R}^3 \). In this space, the distance between two points along the \( i^{th} \) variable is:

\[
d_i(x, x') = \|g_i(x) - g_i(x')\|_2 = \begin{cases} 
0 & \text{if } \delta_i(x) = \delta_i(x') = \text{false} \\
\omega_i & \text{if } \delta_i(x) \neq \delta_i(x') \\
\omega_i \sqrt{2} \sqrt{1 - \cos(\pi \rho_i |x_i - x'_i|)} & \text{if } \delta_i(x) = \delta_i(x') = \text{true}.
\end{cases}
\]

Euclidean distance in the embedded space therefore defines a pseudo-metric, since distinct points can have zero distance. We can use this to define a covariance over our original space. In particular, we consider the class of covariances that are functions only of the Euclidean distance \( \Delta = \|x - x'\|_2 \) between points. There are many examples of such covariances, e.g., the squared exponential and Matérn-\( \frac{3}{2} \) kernels given by Equations (2.5) and (2.6). We can simply take Equation (5.2) in the place of \( \Delta \), returning a valid covariance that satisfies all desiderata above.

Explicitly, note that as desired, if \( i \) is irrelevant for both \( x \) and \( x' \), \( d_i \) specifies that \( g(x) \) and \( g(x') \) should not differ owing to differences between \( x_i \) and \( x'_i \). Secondly, if \( i \) is relevant for both \( x \) and \( x' \), the distance between \( x \) and \( x' \) due to \( x_i \) and \( x'_i \) increases monotonically with increasing \( |x_i - x'_i| \).

The parameter \( \rho_i \) controls whether differing in the relevance of \( i \) contributes more or
less to the distance than differing in the value of \( x_i \), should \( i \) be relevant. In particular, note that for \( \rho_i < \frac{1}{3} \), \( d_i(x, x') < \omega_i \) whenever \( \delta_i(x) = \delta_i(x') = \text{true} \). Conversely, when \( \rho_i > \frac{1}{3} \), \( d_i(x, x') > \omega_i \) whenever \( \delta_i(x) = \delta_i(x') = \text{true} \). Hyperparameter \( \omega_i \) defines a length scale for the \( i \)th feature.

Note that so far we only have defined a kernel for dimension \( i \). To obtain a kernel for the entire \( D \)-dimensional input space, we simply embed each dimension in \( \mathbb{R}^2 \) using Equation (5.1) and then use the embedded input space of size \( 2D \) within any kernel that is defined in terms of Euclidean distance. We refer to this new kernel as the arc kernel, both because it is designed for the architecture search problem and because it measures distance between relevant variables using arc length. Its parameters, \( \omega_i \) and \( \rho_i \) for each dimension, can be optimized using the GP marginal likelihood, or integrated out using MCMC.

5.5 Experiments

Here, we study the effect of the arc kernel on optimization performance. All GP models use a Matérn \( 5/2 \) kernel applied to their respective distance metric.

Data. We use two different datasets. The first is the canonical MNIST digits dataset [LeCun et al., 1998] where the task is to classify handwritten digits. The second is the CIFAR-10 object recognition dataset [Krizhevsky, 2009]. We pre-processed CIFAR-10 by extracting features according to the k-means pipeline given in [Coates et al., 2011].

In this experiment, we test the ability of Bayesian optimization to tune the hyperparameters of each layer of a deep neural network. We allow the neural networks for these problems to use up to 5 hidden layers (or no hidden layer). We optimize over learning rates, L2 weight constraints, dropout rates [Hinton et al., 2012b], and the number of hidden units per layer leading to a total of up to 23 hyperparameters and 6 architectures. On MNIST, most effort is spent improving the error by a fraction of a percent, therefore we optimize this dataset using the log-classification error. For CIFAR-10, we use classification error as the objective. We use the Deepnet\(^1\) package, and each function evaluation took approximately 1000 to 2000 seconds to run on NVIDIA GTX Titan GPUs. Note that when a network of depth \( n \) is tested, all hyperparameters from layers \( n + 1 \) onward are deemed irrelevant.

\(^1\)https://github.com/nitishsrivastava/deepnet
Figure 5.3: Bayesian optimization results using the arc kernel compared to the random embedding baseline.

**Experimental Setup.** For Bayesian optimization, we follow the methodology of [Snoek et al., 2012], using slice sampling and the expected improvement heuristic. In this methodology, the acquisition function is optimized by first selecting from a pre-determined grid of points lying in $[0, 1]^2$, distributed according to a Sobol sequence. We experiment with two baselines. The first is a standard Gaussian process where irrelevant dimensions are projected randomly into the unit hypercube, while the second embeds irrelevant dimensions at 0.

**Results.** Figure 5.3 shows the results against the random embedding baseline. In these cases, Bayesian optimization with the AGP either reaches the best solution faster, or it reaches the a better solution overall. We can see why this might be the case by showing the network depths on the CIFAR-10 experiment in Figure 5.3c. In this case, it is clear
that the AGP explores a variety of network depths, while the random embedding GP picks shallow, less flexible models.

We repeat this experiment, but this time against the zero embedding, and show the results in Figure 5.4. On this, the AGP and baseline perform comparably. Again, we can see why in Figure 5.4c: this time, the baseline tends to favour deeper models. In these neural networks, deeper models are superior, and therefore the inductive bias of the baseline causes it to perform well.

Figure 5.4: Bayesian optimization results using the arc kernel compared to the zero embedding baseline.
5.6 Discussion

For Bayesian optimization, the arc kernel produced mixed results. In terms of coverage, the AGP demonstrated a good ability to explore a wide space of models. For tuning the depth of neural networks, however, this could not outperform the inductive bias of the zero embedding. Essentially, in some neural networks depth is a beneficial property.

We can reason about why this inductive bias exists. Consider that the Euclidean distance used to calculate the kernel is a sum of squared differences along each dimension. For the zero-embedding baseline, shallow networks will have lots of zeros in the hyperparameter space compared to deep networks. This means that the distances between shallow network models will tend to be much smaller than those between deeper networks. The result of this is that the variance around the performance of deeper models will be much higher. This biases expected improvement and causes the exploration to greatly favour deep models. This is exacerbated by the fact that deep models also tend to perform better. In other words, this is a good inductive bias to have for this problem. Notice that the random embedding would not have this bias, and indeed seems to have the opposite bias, where the variance between shallow models is much higher.

For more general conditional spaces, where it is not obvious what the inductive bias should be, the AGP seems like it would be a good choice due to its promotion of exploration. For neural networks specifically, it would be interesting to try to encode this bias more formally in a useful kernel. One caveat is that increasing depth can also result in increased computational cost, which can be taken into account using a cost-sensitive acquisition function, e.g., expected improvement per second [Snoek et al., 2012].

5.7 Conclusion and Future Directions

In this chapter we explored a method for augmenting Gaussian process models to deal with conditional spaces that arise in searches over different architectures. Our solution was to utilize a kernel that is designed to possess some intuitively desirable properties when comparing within and across architectures.

Although in our particular experiments the AGP did not outperform the baseline, it still had the satisfactory property of thoroughly exploring the space of different architectures. Indeed, for more elaborate conditional spaces where the proper inductive bias is not obvious, this can potentially be quite beneficial. For example, there is a much wider range of architectures in the recurrent neural network literature, and it is not always clear which one is superior [Jozefowicz et al., 2015]. In the sense that it is able to promote
exploration, the AGP is a promising step toward proper and robust architecture search in Bayesian optimization.

There are a number of potential directions for future research. To start, the AGP does not satisfactorily deal with what we refer to as the identifiability problem. Namely, the AGP relies on determining a correspondence between the hyperparameters of different architectures. This means that we need a consistent standard for what we select to be the first layer, second layer, and so forth. Consider a one versus a two hidden layer network. In a one layer network we would call the inputs to hidden layer hyperparameters the first set, and the hidden to output hyperparameters the second set. In a two-layer network, the input to hidden could be the first set, but what is the second set? Is it the hidden to hidden hyperparameters? Or is it the hidden to output? Whatever we choose will be establishing an inductive bias that may or may not be appropriate.

An interesting direction for future research would be to consider other techniques for dealing with variable-length inputs. For example, the use of neural networks instead of GPs [Snoek et al., 2015] offers a potential avenue to learn fixed-length embeddings of different architectures. This could also help to avoid the identifiability problem.
Chapter 6

Freeze-Thaw

6.1 Introduction

In machine learning, the term “training” is used to describe the procedure of fitting a model to data. With many models, this fitting procedure is framed as an optimization problem in which a loss is minimized as a function of the parameters. In all but the simplest machine learning models, this minimization must be performed with an iterative algorithm such as stochastic gradient descent. The optimization of hyperparameters therefore usually proceeds as a double loop, where the outer loop selects hyperparameters and the inner loop optimizes the model parameters.

One issue with the typical Bayesian optimization approach is that a model must be fully trained before the quality of its hyperparameters can be assessed. Human experts, however, appear to be able to rapidly assess whether or not a model is likely to eventually be useful, even when the inner-loop training is only partially completed. When such an assessment can be made accurately, it is possible to explore the hyperparameter space more effectively by aborting model fits that are likely to be low quality. The goal of this chapter is to take advantage of the partial information provided by iterative training procedures, within the Bayesian optimization framework for hyperparameter search. We propose a new technique that makes it possible to estimate when to pause the training of one model in favour of starting a new one with different hyperparameters, or to resume the training of a partially-completed model. We refer to our approach as freeze-thaw Bayesian optimization, as the algorithm maintains a set of “frozen” (partially completed but not being actively trained) models and uses an information-theoretic criterion to determine which ones to “thaw” and continue training.

Our approach hinges on the assumption that, for many models, the training loss during the fitting procedure roughly follows an exponential decay towards an unknown
final value. We build a Bayesian nonparametric prior around this assumption by developing a new kernel that is an infinite mixture of exponentially-decaying basis functions, with the goal of characterizing these training curves. Using this kernel with a novel and efficient temporal Gaussian process prior, we are able to forecast the final result of partially trained models and use this during Bayesian optimization to determine the most promising action. We demonstrate that freeze-thaw Bayesian optimization can find good hyperparameter settings for many different models in considerably less time than ordinary Bayesian optimization.

### 6.1.1 A Kernel for Training Curves

We develop here a positive definite covariance kernel designed to model iterative optimization curves. Specifically, we develop a prior that strongly supports exponentially decaying functions of the form $e^{-\lambda t}$ for $t, \lambda \geq 0$. Rather than assume a finite basis with a fixed set of $\lambda$ terms, we integrate over infinite basis functions parameterized by $\lambda$ from 0 to $\infty$ with a mixing measure that allows us to weight regions of the domain. Thus the covariance function $k(t, t')$ between two inputs $t$ and $t'$ is given by:

$$k(t, t') = \int_0^\infty e^{-\lambda t} e^{-\lambda t'} \psi(d\lambda), \quad (6.1)$$

where $\psi$ is a non-negative mixing measure on $\mathbb{R}_+$. It is particularly convenient to take $\psi$ to have the form of a gamma distribution with density $\psi(\lambda) = \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda^{\alpha-1} e^{-\lambda \beta}$, for parameters $\alpha, \beta > 0$ and in which $\Gamma(\cdot)$ is the gamma function. This choice of mixing measure leads to an analytic solution for Equation 6.1:

$$k(t, t') = \frac{\beta^\alpha}{\Gamma(\alpha)} \int_0^\infty e^{-\lambda (t+t')} \lambda^{\alpha-1} e^{-\lambda \beta} d\lambda = \frac{\beta^\alpha}{(t + t' + \beta)\alpha}. \quad (6.2)$$

In Figure 6.1, we show visualizations of the basis set, samples from a Gaussian process prior with this covariance function and samples from a model specifically for optimization curves. The basis set in Figure 6.1a operates over time and is composed of functions of the form $\exp(-\lambda t)$ for varying values of $\lambda$. It is meant to visualize the basis set over which the kernel operates. The basis set has an infinite number of such curves, however $\alpha$ and $\beta$ influences the density of this set. Implicitly, the density is determined by a Gamma distribution over $\lambda$ with shape and rate parameters $\alpha$ and $\beta$ respectively.

In the following, we use this kernel in our model as the covariance function over
time steps for an iterative optimization procedure. For noisy curves, this kernel can be composed with e.g., a noise kernel or an Ornstein-Uhlenbeck kernel. Note that the kernel is invariant to the sign of the basis, meaning that it also captures bases of the form \(-\exp(-\lambda t)\). We don’t find that this poses a problem because if the first observation is greater than the prior mean, then the posterior will put most of its mass on decaying functions.

A similar kernel is developed in [Picheny and Ginsbourger, 2013] by explicitly building desirable properties into a standard kernel using an exponentially decaying amplitude and a change of variables. Interestingly, in that work the assumption is that the observation noise is an asymptotically decreasing function, meaning that there is some true underlying process about which we become more certain over time.
Figure 6.1: Example functions drawn from the exponential decay kernel, where the x-axis represents time. (a) Example basis functions with $\alpha = 1.0$ and $\beta = 0.5$. These basis functions take the form $\exp(-\lambda t)$. The curves are obtained by taking uniformly spaced values between 0 and 1 and then passing them through the inverse CDF of the Gamma distribution, parameterized by $\alpha$ and $\beta$, to get the values of $\lambda$. (b) Samples from a Gaussian process with this covariance function. (c) Samples from a Gaussian process conditioned on the curves starting at a positive number and with an added Ornstein-Uhlenbeck kernel to simulate natural training curves.
6.1.2 An Efficient Hierarchical Gaussian Process

6.1.2.1 Specification

In order to perform Bayesian optimization we need to use a surrogate model as a proxy for the function of interest. The main issue with using GPs is that making predictions requires $O(N^3)$ computation, where $N$ is the number of training observations. In the context of freeze-thaw Bayesian optimization, a naïve model would put a Gaussian process prior over every observed training loss through time. With $N$ unique hyperparameters settings and up to $T$ training iterations per setting, computing quantities of interest using a GP would scale as $O(N^3T^3)$, which is prohibitively expensive.

To reduce the computational cost, we incorporate a conditional independence assumption that each training curve is drawn from a separate Gaussian process, conditioned on the prior mean, which is itself drawn from a global Gaussian process. We define $y_n$ to be a vector of generalization loss measurements for the $n$th hyperparameter setting, i.e., it is a time series. Formally, we model the distribution over these training curves $\{y_n\}_{n=1}^N$ given hyperparameter settings $\{x_n\}_{n=1}^N$ as,

$$
  P(\{y_n\}_{n=1}^N | \{x_n\}_{n=1}^N) = \int \left[ \prod_{n=1}^N \mathcal{N}(y_n; f_n, K_{t_n}) \right] \mathcal{N}(f; m, K_x) \, df,
$$

where $f$ is a latent function that specifies the mean of each training curve, and $1$ is a column vector of ones. In other words, the generative procedure is to first draw a latent function $f$ over hyperparameter settings according to a GP prior. Conditioned on $f$, each training curve is modelled independently using another GP prior with a mean given by the corresponding entry of $f$. We use a constant mean for $m$ (which we infer) and the Matérn-$\frac{5}{2}$ kernel for the GP prior over hyperparameters. Each time-based GP uses the covariance given in Section 6.1.1. A graphical illustration of this model is shown in Figure 6.2a.

The training curves will asymptotically converge to $f$ away from the observed points. As we will demonstrate, the conditional independence assumption is not too restrictive for the kinds of training curves that are typically found when training machine learning models.

Using properties of the Gaussian distribution in Appendix A.3, we can write the joint distribution over $y$ and $f$ as,

$$
  P(y, f | \{x_n\}_{n=1}^N) = \mathcal{N}\left( \begin{pmatrix} f \\ y \end{pmatrix}; \begin{pmatrix} m \\ \text{Om} \end{pmatrix}, \begin{pmatrix} K_x & K_x O^T \\ \text{OK}_x & K_t + \text{OK}_x O^T \end{pmatrix} \right),
$$

(6.4)
where $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N)^\top$, $\mathbf{O} = \text{blockdiag}(\mathbf{1}_1, \mathbf{1}_2, \ldots, \mathbf{1}_N)$ is a block-diagonal matrix, where each block is a vector of ones corresponding to number of observations in its corresponding training curve, and $\mathbf{K}_t = \text{blockdiag}(\mathbf{K}_{t1}, \mathbf{K}_{t2}, \ldots, \mathbf{K}_{tN})$ is a block-diagonal matrix where each block is the covariance for its corresponding training curve.

6.1.2.2 Inference

Using this representation, we can efficiently compute the required quantities for Bayesian optimization. We provide the details below.
**Marginal likelihood** The marginal likelihood is required to estimate the hyperparameters of the GP. Using the marginalization property of the Gaussian distribution in Appendix A.2, the marginal likelihood can be derived from Equation 6.4 and is given by,

\[
P(y \mid \{x_n\}_{n=1}^N) = \mathcal{N}(y; \Omega m, K_t + OK_x O^\top).
\] (6.5)

The covariance of this distribution has a size of \(NT \times NT\), however we can efficiently invert it using the Woodbury matrix identity given in Appendix A.4.

\[
(K_t + OK_x O^\top)^{-1} = K_t^{-1} - K_t^{-1}O(K_x^{-1} + O^\top K_t^{-1}O)^{-1}O^\top K_t^{-1}.
\] (6.6)

We can also use the analogous matrix determinant lemma, given in Appendix A.4, to obtain an efficient representation for the normalization constant of the Gaussian distribution, allowing us to write the log-marginal likelihood as,

\[
\log P(y \mid \{x_n\}_{n=1}^N) = -\frac{1}{2}(y - \Omega m)^\top K_t^{-1}(y - \Omega m) + \frac{1}{2} \gamma^\top (K_x^{-1} + \Lambda)^{-1} \gamma
\]

\[
- \frac{1}{2} \log \left( |K_x^{-1} + \Lambda| \right) + \log \left( |K_x| \right) + \log \left( |K_t| \right) + \text{const}
\] (6.7)

Where \(\gamma = O^\top K_t^{-1}(y - \Omega m)\) and a specific element can be written as, \(\gamma_n = 1_n^\top K_{tn}^{-1}(y_n - m_n)\); \(\Lambda = O^\top K_t^{-1}O = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_N)\), where \(\lambda_n = 1_n^\top K_{tn}^{-1}1_n\).
Posterior distribution

Using the conditioning property of the Gaussian, we can express the posterior \( P(f \mid y, \{x_n\}_{n=1}^N) \) as,

\[
P(f \mid y, \{x_n\}_{n=1}^N) = \mathcal{N}(f; \mu, C),
\]

\[
\mu = m + K_x \gamma - K_x \Lambda (K_x^{-1} + \Lambda)^{-1} \gamma,
\]

\[
= m + (K_x^{-1} + \Lambda)^{-1} \gamma,
\]

\[
= m + C \gamma,
\]

\[
C = K_x - K_x \Lambda K_x + K_x \Lambda (K_x^{-1} + \Lambda)^{-1} \Lambda K_x,
\]

\[
= K_x - K_x (\Lambda - \Lambda (K_x^{-1} + \Lambda)^{-1} \Lambda) K_x,
\]

\[
= (K_x^{-1} + \Lambda)^{-1}.
\]

Posterior predictive distributions

Given a hyperparameter setting \( x_* \), the posterior predictive distribution \( P(f_* \mid y, \{x_n\}_{n=1}^N, x_*) \) is given by,

\[
P(f_* \mid y, \{x_n\}_{n=1}^N, x_*) = \int P(f_* \mid f, x_*) P(f \mid y, \{x_n\}_{n=1}^N) df,
\]

\[
= \int \mathcal{N}(f_*, m + K_{x*}^T K_x^{-1} f, K_{x**} - K_{x*}^T K_x^{-1} K_{x*}) \mathcal{N}(f; \mu, C) df,
\]

\[
= \mathcal{N}(f_*; m + K_{x*}^T K_x^{-1} (\mu - m), K_{x**} - K_{x*}^T K_x^{-1} K_{x*} + K_{x*}^T K_x^{-1} C K_x^{-1} K_{x*}),
\]

\[
= \mathcal{N}(f_*; m + K_{x*}^T K_x^{-1} (\mu - m), K_{x**} - K_{x*}^T (K_x^{-1} - K_x^{-1} C K_x^{-1}) K_{x*}),
\]

\[
= \mathcal{N}(f_*; m + K_{x*}^T K_x^{-1} (\mu - m), K_{x**} - K_{x*}^T (\Lambda - \Lambda C \Lambda) K_{x*}),
\]

\[
= \mathcal{N}(f_*; m + K_{x*}^T K_x^{-1} (\mu - m), K_{x**} - K_{x*}^T (K_x + \Lambda^{-1})^{-1} K_{x*}).
\]

Where \( K_{x**} \) is the test covariance: the input kernel applied to the query point, \( k(x_*, x_*) \),
and \( K_{x*} \) is the cross-covariance between the training inputs and the test input. The posterior predictive distribution for a new point in a training curve, \( y_{n*} \), is given by,

\[
P(y_{n*} \mid \{x_n\}_{n=1}^N, y) = \int P(y_{n*} \mid f_n, \{y_n\}_{n=1}^N) P(f_n \mid y, \{x_n\}_{n=1}^N) df_n,
\]

\[
= \int \mathcal{N}(y_{n*}; f_n 1_* + K_{tn*}^T K_{tn}^{-1} (y_n - 1_n f_n), K_{tn**} - K_{tn*}^T K_{tn}^{-1} K_{tn*}) \mathcal{N}(f_n; \mu_n, C_{nn}) df_n,
\]

\[
= \mathcal{N}(y_{n*}; K_{tn*}^T K_{tn}^{-1} y_n + \Omega \mu_n, K_{tn**} - K_{tn*}^T K_{tn}^{-1} K_{tn*} + \Omega C_{nn} \Omega^T),
\]

\[
\Omega = 1_* - K_{tn*}^T K_{tn}^{-1} 1_n.
\]
Where \(1_*\) is a vector of ones with size equal to the number of time-steps we are predicting. Similar to before, \(K_{tns}\) is the cross-covariance across time between the query point and the rest of the \(n^{th}\) training curve, \(K_{tns^*}\) is the covariance of the query point (the variance for a single query point), and \(K_{tn}\) is the covariance of the \(n^{th}\) training curve observations. Normally we would also need to explicitly refer to \(t\) since the time kernel is non-stationary, however we omit this dependence for brevity since it is a regularly spaced grid in our experiments.

Finally, in the absence of any observations for a particular curve, the posterior predictive distribution is given by,

\[
P(y_* | \{x_n\}_{n=1}^N, y, x_*) = \int P(y_* | f_*) P(f_* | \{x_n\}_{n=1}^N, y, x_*) df_*,
\]

\[
= \int \mathcal{N}(y; f_*, K_{t**}) \mathcal{N}(f_*; \mu_*, \Sigma_{**}) df_*
\]

\[
= \mathcal{N}(y_*; \mu_*, K_{t**} + \Sigma_{**}). \tag{6.13}
\]

Where \(\mu_*\) and \(\Sigma_{**}\) are the mean and variance given by Equation 6.11.

### 6.1.2.3 Computational Complexity

In order to evaluate the GP we need to invert \(K_x\) and \(K_t\) independently. When computing the quantities \(\Lambda\) and \(\gamma\) we can pre-compute the Cholesky decomposition of \(K_t\) and use this to solve linear systems with a \(T \times T\) matrix \(N\) times, leading to a total computational complexity of \(O(N^3 + T^3 + NT^2)\). In practice \(T\) is somewhere between 10 and 100, or can be kept small by sampling the training curves in coarser increments.

### 6.1.3 The Freeze-Thaw Strategy

Using the GP developed in the previous sections, our goal is to create an automatic system that can intelligently decide when to pause training of current models, resume training of previous models, or start new models for training. The optimal strategy is critically dependent on the goal of the user, and we assume that this is to find the best model. That is, if every model were to be fully trained, then the one with the lowest asymptotic error is the one we want to discover. This is reflected in our GP, which becomes a standard GP over hyperparameter settings at the asymptote of each training curve.

Our Bayesian optimization strategy proceeds by maintaining a basket of \(B = B_{\text{old}} + B_{\text{new}}\) candidate models. \(B_{\text{old}}\) represents some number of models that have already been
Algorithm 4 Entropy Search Freeze-Thaw Bayesian Optimization

1: Given a basket \( \{(x, y)\} B_{\text{old}} \cup \{(x)\} B_{\text{new}} \)
2: \( a = (0, 0, \ldots, 0) \)
3: Compute \( P_* \) over the basket using Monte Carlo simulation and Equation 6.11.
4: for each point \( x_k \) in the basket do
5:     // \( n_{\text{fant}} \) is some specified number, e.g., 5.
6:     for \( i = 1 \ldots n_{\text{fant}} \) do
7:         if the point is old then
8:             Fantasize an observation \( y_{t+1} \) using Equation 6.12.
9:             end if
10:        if the point is new then
11:            Fantasize an observation \( y_1 \) using Equation 6.13.
12:       end if
13:       Conditioned on this observation, compute \( P_*^y \) over the basket using Monte Carlo simulation and Equation 6.11.
14:       \( a(k) \leftarrow a(k) + \frac{H(P_*^y) - H(P_*)}{n_{\text{fant}}} \) // information gain.
15:     end for
16: end for
17: Select \( x_k \), where \( k = \arg\max_k a(k) \) as the next model to run.

trained to some degree, while \( B_{\text{new}} \) represents some number of new models. In practice, we set \( B_{\text{old}} = 10 \) and \( B_{\text{new}} = 3 \). The entire basket is chosen using models with the maximum EI at the asymptote, which is computed using Equations (6.11) and (2.9). Each round, after a new observation has been collected, the basket is re-built using possibly different models. This step is essentially standard Bayesian optimization using EI.

Given a basket, the task now becomes one of choosing which candidate to run. Naively choosing EI would always favor picking new models rather than running old ones for more iterations. This is because of the conditional independence assumption of each learning curve. A model trained with some hyperparameter setting for a few iterations will have a lower EI than a model with the same hyperparameters, perturbed by an infinitesimal amount, that has not been trained. This is because the EI of both settings at the asymptote will be extremely close, but the additional uncertainty due to the time-varying component of the untrained model will be higher than the partially trained model.

Instead, similar to the method in Chapter 3, we use the entropy search acquisition function to pick the point that provides the most information about the location of the minimum at the asymptote. Our method is summarized in Algorithm 4. This procedure is similar to standard entropy search as described in Section 2.6.2.4. For each hyperparameter setting in the basket, we simulate the performance obtained by training that model for one additional epoch, or in the case of a new model, training it for a
single epoch, using the posterior distributions in Equations (6.12) and (6.13). We then
treat this as an additional observation for this setting and simulate from the posterior
predictive distribution in Equation (6.11), recording the proportion of times that each
setting in the basket yielded the best result. We can use this to compute the entropy
of the basket, and by extension, the information gain from observing another epoch of
this hyperparameter setting. This procedure is repeated for each hyperparameter setting
in the basket, and the one with the highest information gain (the one that is predicted
to yield the most information about the location of the best hyperparameter setting) is
chosen.

This method does not suffer from the same pathology as expected improvement be-
cause a perturbation of a partially trained model will not necessarily yield more infor-
mation about the location of the minimum after a single epoch. This is because the
uncertainty in the posterior at the asymptote around the partially trained model will
already be reduced due to the observed values of the partially trained model, so mak-
ing highly noisy observations around that point will likely not yield much additional
information.

6.2 Experiments

In this section, we empirically validate our method by comparing to the warped Bayesian
optimization method in Chapter 4 (Warped GP EI MCMC) on four common machine
learning models: online LDA, logistic regression, probabilistic matrix factorization and
convolutional neural networks. For each of these models, we allowed the baseline method
to select the number of training epochs to run, as a hyperparameter to be optimized
between 1 and 100, and report at each epoch the cumulative number of epochs run and
the lowest objective value observed over all epochs on a held-out validation set. We
report our results of the comparison in Figure 6.3, visualizing the problem specific loss
as a function of the total number of training epochs run throughout each of the Bayesian
optimization procedures. Each experiment was run five times and we report the mean
loss. Both methods used input warping to model non-stationarity. Specific details of
our implementation are provided in Appendix C. In all experiments, performance is
measured on a held-out validation set.
6.2.1 Models

Logistic Regression  We optimize five hyperparameters of logistic regression trained using stochastic gradient descent on the popular MNIST data set. The hyperparameters include a norm constraint on the weights (from 0.1 to 20), an \( \ell_2 \) regularization penalty (from 0 to 1), the training minibatch size (from 20 to 2000), dropout regularization [Hinton et al., 2012b] on the training data (from 0 to 0.75) and the learning rate (from \( 10^{-6} \) to \( 10^{-1} \)).

Online LDA  We optimize five hyperparameters of an online latent Dirichlet allocation (LDA) [Hoffman et al., 2010] experiment on 250,000 documents from Wikipedia. We optimize the number of topics (from 2 to 100), two Dirichlet distribution prior base measures (from 0 to 2), and two learning rate parameters (rate from 0.1 to 1, decay from \( 10^{-5} \) to 1). We used the implementation from [Agarwal et al., 2011] and report average perplexity on a held out validation set of 10% of the data.

Probabilistic Matrix Factorization  We optimize three hyperparameters of a probabilistic matrix factorization (PMF) [Salakhutdinov and Mnih, 2008] on 100,000 ratings from the MovieLens data set [Herlocker et al., 1999]. The hyperparameters include the rank (from 0 to 50), the learning rate (from \( 10^{-4} \) to \( 10^{-1} \)) and an \( \ell_2 \) regularization penalty (from 0 to 1).

Convolutional Neural Network  We train a convolutional network using the Tensorflow Cifar-10 convolutional neural net example \(^1\). Using the abbreviation ‘conv’ to denote a convolution layer, ‘fc’ to denote a fully connected layer, ‘pool’ to denote a max-pooling layer, ‘norm’ to denote a local response normalization layer [Krizhevsky et al., 2012], and ‘sm’ to denote a softmax output layer, the architecture is given as conv-pool-norm-conv-norm-pool-fc-fc-sm. We tune five weight-decay settings corresponding to two convolution layers, two fully connected layers, and the softmax output layer. We also tune the global learning rate for an ADAM optimizer [Kingma and Ba, 2015]. Finally, we tune two boolean variables corresponding to keeping or omitting the local response normalization layers. This gives a total of eight hyperparameters. For the purposes of the freeze-thaw algorithm, we define an epoch to be an increment of 1,000 weight updates, or steps, allowing for a total of 100,000 steps per model.

\(^1\)https://www.tensorflow.org/versions/r0.10/tutorials/deep_cnn/index.html
6.2.2 Results

Figure 6.3 shows the results of the empirical analysis in terms of the problem specific loss as a function of the total number of training epochs run out by each method. For the majority of the experiments, our method significantly outperforms traditional Bayesian optimization due to the advantage of being able to dynamically stop and restart experiments. The difference is particularly apparent in the online LDA problem, where we hypothesize that it is relatively easy to predict the shape of the optimization curves given only a small number of observed epochs. We assume that the underlying models being optimized are sufficiently expensive that the cost of fitting the GP is negligible. In practice, the small additional computational effort incurred by our method for explicitly modeling epochs was eclipsed by the performance gains of more rapidly reaching a better loss. For the neural network, the baseline eventually reached the performance and slightly exceeded the freeze-thaw model by a small margin (14.6% ± 0.9% vs 15.2% ± 0.2%). This is likely due to the fact that the freeze-thaw model chose to explore more models, rather than exploiting existing models. Entropy search is known to promote exploration, and it would be useful to investigate alternative acquisition strategies.

In Figure 6.4, we show a visualization of the progression of our Bayesian optimization procedure on the PMF problem. We observe here and throughout the experiments that the method tends to initially explore the hyperparameter space by running only a small number of epochs for various hyperparameter settings. However, once it found a promising curve, it would run it out for more epochs. Later in the optimization, the method would frequently revisit existing curves and extend them for a few epochs at a time, as we observe in Figure 6.4a.

6.3 Conclusion

While training machine learning models with iterative optimization, we constantly receive performance feedback that informs us about how well the model is training. Researchers exploit this information to terminate poor hyperparameter settings early in their optimization in order to rapidly converge on more promising settings. In this chapter we extended Bayesian optimization to also exploit this information by building a model of optimization curves and using this to forecast the eventual model performance. We further provide an entropy search-based strategy that gave the Bayesian optimizer the ability to revive old models that, in light of new information, look more promising than before. In order to accomplish this, we developed a non-stationary kernel that produces
Figure 6.3: This figure shows the results of the empirical comparison to standard Bayesian optimization on four common machine learning hyperparameter optimization problems. For each problem we report the lowest loss observed over all training epochs evaluated by each method, averaged over five optimization runs.

(a) Logistic Regression
(b) Online LDA
(c) PMF
(d) CNN

This work represents a promising first step toward rapid hyperparameter optimization, yet there are many directions for exploration and improvement. The model itself makes two fairly strong assumptions: the shape of the training curve and the conditional independence of different training curves.

One way to deal with the first issue is to use a product kernel of, say, a squared exponential kernel with the exponential decay kernel. This would give the kernel some flexibility to model different shapes of curves, but with the property that it would still
eventually asymptote to some value. Dealing with the second issue is more difficult, as the conditional independence assumption leads to computational efficiency. Still, sparse GP methods [Quinonero-Candela and Rasmussen, 2005] may provide a way to share information while keeping the computational cost reasonable.

Although we built in the ability to revive old models, this may not be necessary in practice. Indeed, if the goal is to find the best model then it may be better to explore a new hyperparameter setting rather than returning to an old one. If we remove this feature, then we can use much simpler acquisition functions. An example is given in [Domhan et al., 2014], where they simply terminate a model if it is forecasted to be worse than the current best with some probability \( p \), e.g., \( p > 0.99 \).

Exploiting partial information should not be thought of as an all-or-none solution. A simple way to utilize this information in an existing Bayesian optimization scheme is to allow researchers to periodically inspect the learning curves. Those that are deemed unpromising can then be terminated manually. The downside is that this takes human intervention and specialized software that is capable of interaction. It would also be important to discourage the optimizer from returning to these hyperparameter regions. This could be done with e.g., constraints [Gelbart et al., 2014].

Finally, while we have considered the hyperparameters in each model to be fixed
throughout the training duration, an interesting direction would be to start with a single model and adapt the hyperparameters over time. An example of this for learning rates is given in [Bache et al., 2014].
Chapter 7

Conclusion

Hyperparameter optimization is a problem that appears in nearly every machine learning application—in particular, deep learning. Deep learning architectures have been progressively becoming more complicated and much deeper, sometimes involving up to one thousand layers [He et al., 2015]. The space of nonlinearities is becoming much richer, and much more heavily parameterized [Goodfellow et al., 2013, Clevert et al., 2015]. New mechanisms such as attention are now being used in applications such as caption generation [Xu et al., 2015], image generation [Gregor et al., 2015] and neural computation [Graves et al., 2014]. The field of recurrent neural networks has become inundated with various forms of gating mechanisms [Hochreiter and Schmidhuber, 1997, Cho et al., 2014]. There is even an emerging field of deep learning inspired art [Gatys et al., 2015]. All of these extensions introduce new hyperparameters to tune, many of which do not have a rich body of prior work on which to develop heuristics for setting them. The space of deep learning models is growing rapidly. In light of this, an automatic, systematic process for optimizing hyperparameters is appealing because it allows researchers to spend their time exploring new models, rather than tweaking old ones. It also provides a degree of reproducability and removes bias across researchers that is essential for determining which models are genuinely good.

Bayesian optimization is a promising approach toward this goal in that it has been shown to be a good and robust black-box approach for many hyperparameter optimization problems, while requiring very little prior information. However, this strength becomes a weakness as models become more complex. In order to simultaneously learn and optimize the objective function without an informative prior, Bayesian optimization must query the function many times. This can become prohibitively expensive. Researchers solve this problem by taking the time to understand how the hyperparameters of a model affect the final outcome. They build this knowledge over time, often by ap-
plying the same model to multiple datasets. This is extremely difficult in even modestly high dimensions, but it establishes a reasonable prior that makes subsequent work with the model far easier. In this thesis, we took inspiration from this ability and developed ways to build similar priors into the Bayesian optimization framework. In particular, we focused on four key types of priors.

The first prior is that the same model applied to similar datasets will behave similarly across hyperparameter settings. Another way to think about this is that we can augment the size of our dataset by borrowing information from related problems. This kind of problem is known in machine learning as multi-task learning, and using multi-task Gaussian process models, we were able to build this prior into Bayesian optimization. The result is that the model requires far fewer examples on the current task in order to learn and optimize the objective, resulting in greater efficiency. Of particular interest is the ability to heavily explore a small, cheap problem and use this to rapidly solve a related, expensive problem.

The second prior deals with the non-uniform sensitivity of the model to its hyperparameters. Researchers deal with this problem by reasoning about parameterizations of the hyperparameter space. We formalized this as a problem of non-stationarity and developed a novel warping technique to detect and remove these non-stationary effects. This introduces very little overhead in the existing Bayesian optimization framework, but provides a pronounced gain in speed and robustness. It also provides some insight, as researchers can inspect the learned transformations.

The third prior involves discovering the best model architecture. This involves conditional hyperparameter spaces, where the relevance of some hyperparameters to the model depends on the values of other hyperparameters. Researchers are able to reason about the relationships between different architectures and can use this to discover good architectures without exhaustively exploring each one. We investigated a particular approach to augmenting Bayesian optimization with the capability of handling conditional hyperparameters based on some intuitive criteria of how different architectures should relate to each other. The crux of this approach was to embed a variable-sized input (the conditional space) in a fixed-sized vector. In a sense, this also transforms the problem of architecture search into a multi-task problem, where each architecture is a separate task.

The fourth, and final prior, builds on the ability of researchers to open the black-box and investigate a model as it is training. Traditional Bayesian optimization requires that an experiment be run to completion before its information can be added to the knowledge base of the surrogate model. Researchers have realized, however, that optimization curves (particularly well-behaved ones) tend to follow very predictable patterns; usually
Chapter 7. Conclusion

We used this intuition to develop a prior that can forecast training curves given partially trained models. Our approach can decide which models to keep training, which ones to stop training, and also which models to bring back and resume training. The ability to terminate a poorly performing model is a desirable feature that can yield substantial improvements in optimization speed.

Together, these techniques help to overcome the shortcomings of the traditional Bayesian optimization approach. By basing these extensions on the abilities of researchers, we have created a system that unifies the strengths of both systematic optimization and human intuition.

There are many directions in which this can be taken in the future. Perhaps the most pressing issue is the scalability of Gaussian process models. In the work we presented here, this was overcome by making restrictive assumptions, such as limiting the number of tasks or making conditional independence assumptions. A truly versatile system must be capable of using a vast amount of data from prior tasks, and must be able to execute many queries in parallel. Some hyperparameter tuning systems are much more scalable compared to GPs [Bergstra et al., 2011, Hutter et al., 2010], but they lack the principled Bayesian foundation on which to build in formal priors. Other work has focused on sparse Gaussian process models [Nickson et al., 2014]. Recent work has investigated replacing the GP with a Bayesian neural network [Snoek et al., 2015]. Indeed, there is now a resurgence in Bayesian neural networks thanks to new techniques in black-box variational inference [Ranganath et al., 2013, Kingma and Welling, 2014, Rezende et al., 2014, Blundell et al., 2015].

Another issue is dimensionality. Standard Gaussian process kernels suffer from the curse of dimensionality due to their use of the Euclidean distance metric. The practical effect is that in high dimensions, they will require many more observations to accurately model the objective function. There have been attempts to alleviate this issue by discovering a low-dimensional embedding using random projections [Wang et al., 2013]. The use of neural networks, as in [Snoek et al., 2015] could also potentially be helpful here in discovering a nonlinear embedding. Another direction would be to use gradient information, which can be readily incorporated into Gaussian process models. An example of computing the gradient of the validation error with respect to continuous hyperparameters is given in [Maclaurin et al., 2015].

Although the focus in this thesis has been on hyperparameter optimization, the world in which Bayesian optimization can be applied is far more vast. Examples abound already in applications like gait optimization in bipedal robots [Calandra et al., 2014], designing molecules to possess desirable properties [Duvenaud et al., 2015], optimizing turbines for
greater efficiency [Talnikar et al., 2014], selecting sensors in sensor networks [Garnett et al., 2010], designing graphics and visual effects [Brochu et al., 2010a], and more. Hyperparameter tuning is a particularly useful domain to work with because it is almost entirely software based and requires little to no human intervention once the model and data have been given. It is an excellent domain in which to develop extensions, like those found in this thesis, that can then be applied in other domains.

The primary goal of this thesis was to use prior information to increase the speed and robustness of Bayesian optimization, to allow it to scale to difficult, real-world machine learning problems. An orthogonal direction is to then use Bayesian optimization as a tool to better understand machine learning and deep learning systems. For example: how do the best hyperparameters of a model vary across datasets? What architectures tend to be most effective in which scenarios? Do different models behave similarly with respect to their hyperparameters? How many modes are there in the hyperparameter space, and how difficult are they to find? Developing techniques to answer these questions would yield valuable insight into machine learning and deep learning systems.
Chapter 8

Appendix

A Multivariate Gaussian Distribution Identities

A.1 Density

The multivariate Gaussian distribution is a distribution over variables $\mathbf{x} \in \mathbb{R}^D$. It is specified by a mean $\mu$ and covariance $\Sigma$. The density is given by:

$$P(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, \Sigma) = (2\pi)^{-D/2} |\Sigma|^{-1/2} \exp \left( -\frac{1}{2} (\mathbf{x} - \mu)^\top \Sigma^{-1} (\mathbf{x} - \mu) \right).$$

A.2 Marginalization and Conditioning

The Gaussian distribution is closed under marginalization and conditioning. For two random Gaussian vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ with a joint multivariate Gaussian distribution given by,

$$P(\mathbf{x}_1, \mathbf{x}_2) = \mathcal{N} \left( \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}; \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21}^T & \Sigma_{22} \end{pmatrix} \right),$$

we can write the marginal and conditional distributions of $\mathbf{x}_1$ as,

$$P(\mathbf{x}_1) = \mathcal{N}(\mathbf{x}_1; \mu_1, \Sigma_{11}),$$
$$P(\mathbf{x}_1|\mathbf{x}_2) = \mathcal{N}(\mathbf{x}_1; \mu_1 + \Sigma_{12} \Sigma_{22}^{-1}(\mathbf{x}_2 - \mu_2), \Sigma_{11} - \Sigma_{12} \Sigma_{22}^{-1} \Sigma_{21}^T).$$
A.3 Affine Transformations

Suppose $x_1$ and $x_2$ are distributed as follows:

$$P(x_1) = \mathcal{N}(x_1; \mu_1, \Sigma_{11})$$
$$P(x_2|x_1) = \mathcal{N}(x_2; Ax_1, \Sigma_{22}).$$

For some matrix $A$. We can write their joint distribution as,

$$P(x_1, x_2) = \mathcal{N}\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}; \begin{pmatrix} \mu_1 \\ A\mu_1 \end{pmatrix}, \begin{pmatrix} \Sigma_{11} & \Sigma_{11}A^T \\ A\Sigma_{11} & \Sigma_{22} + A\Sigma_{11}A^T \end{pmatrix}\right).$$

From this, we can apply the identities in Section A.2 to get the corresponding marginal and conditional distributions.

A.4 Matrix Inversion and Determinant Lemmas

The matrix inversion lemma [Petersen et al., 2008] allows us to invert a diagonal plus $N \times N$ matrix with rank $K$ in $O(NK^2)$ time rather than $O(N^3)$, which can be much faster when $K << N$.

Given a diagonal matrix $D$, and a low-rank matrix $M$ of the form $M = ACB$ where $C$ is a $K \times K$ matrix, the matrix inversion lemma allows us to write the inverse of $D + M$ as follows:

$$(D + M)^{-1} = (D + ACB)^{-1},$$
$$= D^{-1} - D^{-1}A(C^{-1} + BD^{-1}A)^{-1}BD^{-1}.$$

The analogous matrix determinant lemma [Petersen et al., 2008] allows us to efficiently compute the matrix determinant:

$$|D + M| = |D + ACB|$$
$$= |C^{-1} + BD^{-1}A||C||D|.$$
B Multi-Task Experiment Details

B.1 Cold Start Experiments

Logistic regression

This experiment uses a simple logistic regression classifier taking the pixels as input and is trained with stochastic gradient descent.

Best hyperparameter settings

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Learning rate</th>
<th>$\ell_2$ penalty</th>
<th>Batch size</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>USPS</td>
<td>0.0002</td>
<td>0.0032</td>
<td>560</td>
<td>161</td>
</tr>
<tr>
<td>MNIST</td>
<td>0.1435</td>
<td>0.0</td>
<td>206</td>
<td>685</td>
</tr>
</tbody>
</table>

Cuda-Convnet

We used the default architecture from the Cuda-Convnet package. This is a network that consists of two convolutional layers and two densely connected layers. When performing the transfer experiment, the epochs are scaled so that the same number of weight updates are applied to the model for both datasets. In this case, 18 epochs on SVHN is equivalent to 450 epochs on CIFAR-10. The small version of SVHN does not scale epochs, so it is only allowed to use up to 10% of the weight updates of the other models. The correlation between SVHN and CIFAR-10 was measured at $0.59 \pm 0.22$ while the correlation between SVHN and its small counterpart was measured at $0.62 \pm 0.24$.

More information on this architecture can be found on the package website:

Best hyperparameter settings (continued)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Scale</th>
<th>Pow</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>3</td>
<td>0.0480</td>
<td>0.2136</td>
<td>321</td>
</tr>
<tr>
<td>SVHN</td>
<td>3</td>
<td>0.0788</td>
<td>0.2700</td>
<td>18</td>
</tr>
<tr>
<td>SVHN (small)</td>
<td>2</td>
<td>0.1</td>
<td>0.0902</td>
<td>18</td>
</tr>
</tbody>
</table>
Deepnet on k-means features
We first extract $n$ k-means features, where $n \in \{400, 1000\}$, from whitened image patches extracted from the STL-10 unsupervised image set. These are then combined using max-pooling into a $m \times m$ grid, where $m \in \{5, 7, 9\}$ resulting in a $m \times m \times n$ set of features. A convolutional neural network containing one convolutional hidden layer and one densely connected hidden layer is then applied to these. In this case, $n$ would be analogous to color channels. Each trial was allowed to use 100000 weight updates with a batch size of 128 and the final set of weights are used for classification. The correlation between datasets was measured at $0.5 \pm 0.28$.

The Deepnet package can be found at

https://github.com/nitishsrivastava/deepnet and the full model specifications will be posted on the authors website.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Learning rates</th>
<th>Max pooling grid size</th>
<th>Number of k-means features</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>0.0031, 0.0034, 0.0007</td>
<td>9x9</td>
<td>400</td>
</tr>
<tr>
<td>STL-10</td>
<td>0.1, 0.1, 1e-5</td>
<td>7x7</td>
<td>1000</td>
</tr>
</tbody>
</table>

Best hyperparameter settings (continued)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of hidden units</th>
<th>Weight norm constraints</th>
<th>Dropout probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>1000, 2000</td>
<td>0.25, 8.0, 2.441</td>
<td>0.7035, 0.1955, 0.4915</td>
</tr>
<tr>
<td>STL-10</td>
<td>2000, 1100</td>
<td>0.25, 3.221, 0.25</td>
<td>0.5925, 0.7185, 0.9</td>
</tr>
</tbody>
</table>

C Freeze-Thaw Implementation Details

In our experiments we follow the conventions laid out in [Snoek et al., 2012] and use a modification of it’s accompanying Spearmint package \(^1\). The specific details of our GP model implementation are given below.

\(^1\)https://github.com/JasperSnoek/spearmint
C.1 Kernel and GP Hyperparameters

We use a Matérn-$\frac{5}{2}$ to determine the function over hyperparameters, along with the warping technique developed in Chapter 4.

\[
k_{M52}(w(x), w(x')) = \theta_0 \left( 1 + \sqrt{5r^2} + \frac{5}{3}r^2 \right) \exp \left( -\sqrt{5r^2} \right),
\]

\[
r^2 = \sum_{d=1}^{D} \frac{(w_d(x) - w_d(x'))^2}{\theta_d^2},
\]

\[
w_d(x) = \text{BetaCDF}(x, a_d, b_d).
\]

Where BetaCDF refers to the cumulative distribution of the beta distribution with shape parameters $a$ and $b$. We place a log-normal prior with a log-scale of 0 on $\theta_0$, $a_d$, and $b_d$ for all $d$, and a top-hat prior on $\theta_d$,

\[
\theta_0 \sim \text{lognorm}(0, 1),
\]

\[
a_d \sim \text{lognorm}(0, 1) \quad d = 1...D,
\]

\[
b_d \sim \text{lognorm}(0, 1) \quad d = 1...D,
\]

\[
\theta_d \sim \text{uniform}(0, 10).
\]

For the kernel over epochs, we use our custom exponential decay kernel along with an additive noise kernel.

\[
k_{\text{exp decay}}(t, t') = \frac{\beta^\alpha}{(t + t' + \beta)^\alpha} + \delta(t, t')\sigma^2.
\]

We place a lognormal prior on the hyperparameters $\alpha$ and $\beta$, and a horseshoe prior on $\sigma^2$.

\[
\alpha \sim \text{lognorm}(0, 1) \quad d = 1...D,
\]

\[
\beta \sim \text{lognorm}(0, 1) \quad d = 1...D,
\]

\[
\sigma^2 \sim \log(\log(1 + \frac{0.1}{\sigma^2})).
\]

Finally, we use a constant prior mean $m$ with a uniform hyperprior for the GP over hyperparameters. We ensure that this value does not exceed the bounds of the observations.

\[
m \sim \text{uniform}(y_{\min}, y_{\max}).
\]
Bibliography


