Training and Inference for Deep Gaussian Processes

Keyon Vafa

advised by
Alexander Rush

A thesis submitted in partial fulfillment of the requirements
for the joint degree of
Bachelor of Arts
in Computer Science and Statistics

Harvard College
April 1, 2016

Abstract

An ideal model for regression is not only accurate, but also computationally efficient, easy to tune without overfitting, and able to provide certainty estimates. In this thesis, we explore deep Gaussian processes (deep GPs), a class of models for regression that combines Gaussian processes (GPs) with deep architectures. Exact inference on deep GPs is intractable, and while researchers have proposed variational approximation methods, these models are difficult to implement and do not extend easily to arbitrary kernels. In this thesis, we introduce the Deep Gaussian Process Sampling algorithm (DGPS), which relies on Monte Carlo sampling to circumvent the intractability hurdle and uses pseudo data to ease the computational burden. We build the intuition for this algorithm by defining and discussing GPs and deep GPs, going over their strengths and limitations as models. We then apply the DGPS algorithm to various data sets, and show that deeper architectures are better suited than single-layer GPs to learn complicated functions, especially those involving non-stationary data, although training becomes more difficult due to limitations of local maxima. Throughout, our goal is not only to introduce a novel inference technique, but also to make deep Gaussian processes more accessible to the machine learning community at large.

Acknowledgements

First off, I would like to thank my advisor, Sasha Rush, for his incredible mentorship throughout the thesis process. Last summer, when I discovered Sasha would be joining Harvard's computer science faculty, I cold-emailed him, asking him to meet to discuss potential thesis topics. He responded almost instantly, arranging a meeting where he and his graduate students walked me through their research, carefully explaining all the details that went over my head. This thesis would not be possible without his guidance, enthusiasm, and patience over the past seven months.

Additionally, Finale Doshi-Velez, David Duvenaud, and Miguel Hernández-Lobato have been instrumental mentors. Finale first introduced me to deep Gaussian processes, and has been eager to clarify all my questions and discover real-world applications of this research. Meanwhile, David and Miguel were there every step of the way both to discuss the high-level modeling ideas and to get into the nitty-gritty parts of the code. They always eagerly responded to my emails sent at 3 in the morning, and the Deep Gaussian Process Sampling algorithm would not exist without them.

Many people gave me valuable advice as I was looking for thesis topics and developing ideas. Joe Blitzstein has been an outstanding mentor throughout my time at Harvard – I took my first statistics class with Joe, and he provided crucial high-level advice for my thesis. Next time, Chipotle's on me! Ryan Adams and Natesh Pillai helped me explore thesis topics early on and gave great insight when I had no idea what I was doing. Edo Airoldi has been there to bounce off ideas throughout this process, and he has offered firm support and guidance throughout my time at Harvard. Guillaume Pouliot was always helpful when I got stuck, and his passion for statistics has motivated me to explore further research. David Donoho, although 3,000 miles away, managed to offer fundamental advice and make sure I stay on track with my research and writing, and Peter Brown has similarly been eager to pick up the phone whenever I had questions about statistics or computer science. I would also like to thank David Parkes, not only for helping me brainstorm thesis topics, but for

giving me an extension on my CS 136 problem set the day this thesis was due.

My friends have offered me unparalleled support throughout my time spent on this thesis and in school. I would like to thank Nathan Daniel and Ben Scharfstein, whom I have known since my first year of high school, for their friendship over the past eight years. I would also like to thank my other blockmates and roommates: Isaac Inkeles, Alex Kiam, Teddy Kim, Alex Hem, Hank Sneddon, James Lim, Camila Victoriano, Kia Turner, and Savanna Arral. Additionally, I would like to thank Mark Arildsen, Ashley Bae, and Katherine Chen for their friendship and for their invaluable feedback on my rough drafts.

I would like to thank Kathy Li for her endless support and for her unbounded enthusiasm. Writing a thesis is a marathon, not a sprint, and because of Kathy, it has been exciting and enjoyable every step of the way.

Finally, I would like to thank my family – Mom, Dad, Farzan, and Neekon. Without your love, guidance, and inspiration, none of this would be possible. This thesis is dedicated to you.

Contents

Chapter 1

Introduction

1.1 Overview

The ability to make predictions is central to machine learning. As the size of recorded data in the world increases and researchers become more data-aware, we have seen numerous successful implementations of prediction models. Nate Silver can forecast the outcome of the next election. Biologists can predict DNA-binding proteins. Facebook can identify your friends in pictures.

Many predictions today are made using supervised learning techniques. That is, a particular prediction model "trains" on a labelled data set, extrapolating knowledge about the data's structure to make predictions for unlabelled data. This framework provides a way to evaluate the success of these models – not only can we observe how our model adapts to the training data, but we can also evaluate how closely our predictions match the actual results.

It is hard to define the criteria that make a prediction model *good.* Accuracy is perhaps the most important characteristic of a successful method, as an ideal model will make perfect predictions. However, it is far from a sufficient condition. What if the computational complexity of training is infeasible? Can we interpret the results of our model? In this thesis, we focus on three desiderata for prediction models: we prefer models that are computationally efficient, easy to tune without overfitting, and able to provide certainty estimates.

Many present-day predictions are performed on large datasets, both in the number of data points and in the dimension of each input. Thus, a successful algorithm must scale well when training and making predictions. Oftentimes, perfect inference is not computationally feasible, so

certain models rely on approximations of the data. As a result, a successful model should not incur a dramatic loss in accuracy due to any underlying approximations.

Additionally, every family of models can depend on a different set of model parameters, or hyper-parameters, that need to be tuned to produce optimal predictions. For example, if we would like to cluster our data into groups before making predictions, a model parameter would be the number of groups to use. While techniques like cross-validation exist, where the training data is used to evaluate the predictive accuracy for each combination of parameters, it can be expensive to test every combination. Thus, it is desirable to evaluate the likelihood of a particular model in order to choose the optimal set of hyper-parameters, a task known as model selection. A closely related challenge is overfitting. When we train an algorithm on training data, we risk fitting the data so closely that our test set predictions are inaccurate. Frequently, this occurs when we are trying to predict a large number of parameter values, resulting in a fit that can be very exact on our training data but offer poor predictive performance. Thus, a successful prediction method should be able to guard against overfitting.

Finally, our model should give us a sense of how certain we are of our estimates. There are many advantages of having an idea of our certainty instead of producing a single point-estimate, as many black-box methods do. Frequently, we are not concerned with the most likely outcome of events, but instead the range of likelihoods. For example, if we would like to model the outcome of a presidential primary, we oftentimes want to know more than who will win. How certain are we of the presumed winner? What probabilities do each of the other candidates have of winning? What is the distribution of delegates?

This thesis will focus on one particular class of prediction models: deep Gaussian processes for regression. There are many reasons to study deep Gaussian processes (deep GPs). For one, they are a relatively new class of models, having been introduced in 2013. Thus, there are numerous properties and open questions to explore, many of them related to learning and inference.

While deep GPs are a well-defined model, exact inference is intractable. In this thesis, we will introduce a new method to fit deep GPs, which we call the Deep Gaussian Process Sampling algorithm, or DGPS for short. We will argue that the DGPS algorithm is more straightforward than alternative methods for deep Gaussian process inference, and that it can more easily adapt to arbitrary data structures. Additionally, we will argue that training deep GPs with the DGPS

algorithm fulfills the three outlined desiderata for prediction models: it is computationally efficient, easy to tune without overfitting, and able to provide certainty estimates.

In Chapter 2, we begin by discussing Gaussian processes (GPs), the building blocks of deep GPs, from a mathematical perspective. GPs provide a distribution over functions, and in the context of regression, we can train a GP to fit our training data. Additionally, the Bayesian nature of GPs provides an efficient method for model selection, as it guards against overfitting.

In Chapter 3, we introduce deep Gaussian processes. Their deep architecture, which consists of layers, where each unit in a given layer corresponds to a GP, is inspired by that of neural networks. Moreover, we show how deep GPs and neural networks are closely intertwined. We then highlight a significant advantage of deep GPs over single-layer GPs: that they can more easily adapt to learn non-stationary functions.

In Chapter 4, we show that exact inference, which relies on integrating out hidden function values, is intractable. We introduce the DPGS algorithm, which uses Monte Carlo sampling to overcome the intractability hurdle. Additionally, it relies on an approximation involving pseudo data to ease the computational burden. We discuss the benefits of this method, namely that it is straightforward to implement and easily extendible across data sets. We then walk through an example and provide pseudocode for general architectures.

In Chapter 5, we use the DGPS algorithm to fit training data, and evaluate the success of our predictions. Using both non-stationary toy data and real-world data, we will show how incorporating additional layers into our models can improve predictions. We also note the computational challenges of adding more layers – as the number of layers and model parameters grows, we are more likely to encounter local optima during training. We use experiments and empirical findings to back these results.

In Chapter 6, we discuss suggestions for improving the DGPS algorithm. These involve extending the DGPS algorithm to classification, experimenting with alternative optimization techniques, and evaluating the model likelihood to choose the model with the optimal architecture.

Deep Gaussian processes are exciting models, and they have the potential to produce high-quality predictions in many applied scenarios. Through this thesis, our goal is two-fold. For one, we intend to contribute to ongoing research on deep GPs by introducing the Deep Gaussian Process Sampling algorithm. Additionally, much of the current literature on deep GPs is written for

advanced audiences with extensive backgrounds in Bayesian inference. By developing an intuition for deep GPs and discussing in depth their strength as a model – along with introducing the DGPS algorithm, which is more straightforward than existing inference methods for deep GPs – we aim to make deep Gaussian processes more accessible to the machine learning community at large.

1.2 Notation

In order to be consistent with notation throughout this thesis, we will introduce the main notational elements here.

$\mathbb{R}^D$ corresponds to the D-dimensional vector space of reals. We will differentiate between scalars, vectors, and matrices by bolding anything that is more than 1-dimensional, and capitalizing matrix names. For example, x is a scalar, while $\boldsymbol{x} \in \mathbb{R}^D$ is a vector, and $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ is a matrix. We denote the transpose of a matrix $\boldsymbol{X}$ as $\boldsymbol{X}^T$. Additionally, $\boldsymbol{I}$ corresponds to the Identity matrix, and its dimensionality should be clear from context.

We will use $\mathcal{N}(\mu, \sigma^2)$ to represent a normal (Gaussian) distribution with mean μ and variance σ^2, where μ and σ^2 are both scalars, with σ^2 positive. We will also use $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ to represent a multivariate normal distribution of dimension K if $\boldsymbol{\mu} \in \mathbb{R}^K$ and $\boldsymbol{\Sigma} \in \mathbb{R}^{K \times K}$, with $\boldsymbol{\Sigma}$ positive-definite. To avoid confusion, the dimensionality of the Gaussian should be clear from context and from whether the parameters are bolded.

In a regression setting, we will always be dealing with data (or inputs) $\{\boldsymbol{x}_n\}_{n=1}^N \in \mathbb{R}^D$, where N is the number of data points, and D the dimensionality of the data. Each element $\boldsymbol{x}_n$ will have a label $y_n \in \mathbb{R}$. The matrix generated by stacking the row-vectors for every data point will be notated as $\boldsymbol{X}$, and the 1-dimensional vector of labels is $\boldsymbol{y}$. Note that $\boldsymbol{X}$ may be $N \times 1$ when the data $x_n \in \boldsymbol{X}$ is a scalar. In the context of a deep GP, the dimensionality of $\boldsymbol{x}_n$ will be $D^{(0)}$, the input dimension, so as not to confuse it with the dimensionality of the other layers.

Oftentimes, we would like to evaluate a kernel matrix with respect to a kernel $k(\boldsymbol{x}, \boldsymbol{x}')$ at values $\boldsymbol{X}$ and $\boldsymbol{X}'$. Thus, $\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}'}$ refers to the matrix where the (i, j) element is $k(\boldsymbol{x}_i, \boldsymbol{x}'_j)$. Even when the kernel takes as input two scalars and $\boldsymbol{X} \in \mathbb{R}^{D \times 1}$, we still represent the kernel matrix as $\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}'}$, except now the (i, j) element is $k(x_i, x'_j)$.

When we use the $\log(\cdot)$ function, it is referring to the logarithm of our input in the natural

base, e.

Finally, we will abbreviate "independent and identically distributed" to "i.i.d."

Chapter 2

Gaussian Processes

In this chapter, we introduce Gaussian processes (or GPs), the building blocks of a deep Gaussian process. Intuitively, a GP is the infinite-dimensional extension of a multivariate normal, and so it can be thought of as a function, indexed by input values and their corresponding outputs. This gives us a new way to envision data; given N observations in a data set, we can imagine the outputs as a single vector sampled from an N-dimensional multivariate normal. Thus, the goal in regression is to predict the function values at new inputs, conditioning on our observations. A GP is considered a *nonparametric* model, as the size of the model grows as the number of observations increases. The probabilistic and Bayesian nature of a GP has many advantages: we can evaluate the marginal likelihood of our model, making model selection a numerically tractable task. Additionally, overfitting is seldom a problem due to the lack of parameters that need to be estimated.

We will begin by discussing the background and motivation for GPs, reviewing the multivariate normal distribution. We then focus specifically on fitting a GP for regression. Finally, we conclude by discussing the advantages of using a GP for prediction.

2.1 Multivariate Normal Distribution

We begin by reviewing the multivariate normal distribution, which is central to understanding Gaussian processes. A random vector $\boldsymbol{y} = (y_1, \ldots, y_K)$ has the multivariate normal distribution if it is the following transformation of $\boldsymbol{z} = (z_1, \ldots, z_M)$, where each $z_m \sim \mathcal{N}(0, 1)$ i.i.d (Blitzstein and

Morris, 2016):

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{z} + \boldsymbol{\mu}$$

where $\boldsymbol{A}$ is a $K \times M$ matrix and $\boldsymbol{\mu} \in \mathbb{R}^K$. We represent this as $\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma} = \text{Cov}(\boldsymbol{y}, \boldsymbol{y}) = \boldsymbol{A}\boldsymbol{A}^T$. $\boldsymbol{\mu}$ is referred to as the mean vector, and $\boldsymbol{\Sigma}$ the covariance matrix. The probability density function is given by

$$f(\boldsymbol{y}) = \frac{1}{(2\pi)^{K/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{y}-\boldsymbol{\mu})\right). \tag{2.1}$$

In the context of Gaussian processes for regression, perhaps the most important characteristic of the multivariate normal is the conditional distribution. We will later show that GP regression corresponds to the task of estimating the values of a multivariate normal vector at new inputs; therefore, it is imperative to understand the distribution of unobserved data conditioned on observed data. Consider a multivariate normal variable $\boldsymbol{y}$ split up into two sub-vectors, $(\boldsymbol{y}_1, \boldsymbol{y}_2)$. Each of these sub-vectors has a multivariate normal distribution – this is true because we can split up $\boldsymbol{A}$ in the representation above to the rows that correspond to $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ to generate two new matrices. We can write $\boldsymbol{y}$ in terms of its two sub-vectors,

$$\boldsymbol{y} = \begin{pmatrix}\boldsymbol{y}_1 \\ \boldsymbol{y}_2\end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix}\boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2\end{pmatrix}, \begin{pmatrix}\boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22}\end{pmatrix}\right).$$

We are interested in the conditional distribution of $\boldsymbol{y}_2$ given $\boldsymbol{y}_1$. Using linear algebra techniques (Blitzstein and Morris, 2016), we can derive $P(\boldsymbol{y}_1|\boldsymbol{y}_2) = \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$ where

$$\boldsymbol{\mu}_* = \boldsymbol{\mu}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\boldsymbol{y}_1 - \boldsymbol{\mu}_1) \text{ and } \boldsymbol{\Sigma}_* = \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}. \tag{2.2}$$

2.2 Definition

Intuitively, we can think of Gaussian processes as the infinite dimensional extension of a multivariate normal distribution. Thus, as an infinite collection of random variables, a Gaussian process is a function. Formally, a function $\boldsymbol{f}$ is a Gaussian process if any finite set of values $f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N)$

has a multivariate normal distribution, where the inputs $\{\boldsymbol{x}_n\}_{n=1}^N$ correspond to vectors from any arbitrary sized domain (Rasmussen and Williams, 2006). We usually denote a GP in terms of its inputs $f(\boldsymbol{x}_n)$, but occasionally we shorten it to $\boldsymbol{f}$ to represent the function vector evaluated at all inputs. Rather than being specified by parameters, a GP is specified by a mean function $m(\boldsymbol{x})$ and a covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$, which we denote as $f \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}'))$. That is, for any $\boldsymbol{x}, \boldsymbol{x}'$:

$$m(\boldsymbol{x}) = E[f(\boldsymbol{x})]$$

and

$$k(\boldsymbol{x}, \boldsymbol{x}') = \text{Cov}(f(\boldsymbol{x}), f(\boldsymbol{x}')).$$

A Gaussian process is a *nonparametric* model, meaning our parameter-space is infinite dimensional. That is, our function does not depend on fixed parameters – rather, the size of our model grows as the number of data points increases (Orbanz and Teh, 2011). The shape of our function is determined by the covariance function, as it controls the dependence of all pairs of output values, $f(\boldsymbol{x})$ and $f(\boldsymbol{x}')$. As a result, the covariance function can determine the smoothness and stationarity of the function (Rasmussen and Williams, 2006). We can usually incorporate knowledge about our mean into the covariance function, so we typically assume a GP to have zero mean. (Duvenaud, 2014)

2.3 Kernels

The only requirement of a covariance function, also known as a kernel, is that it be a positive-definite function that maps two inputs, $\boldsymbol{x}$ and $\boldsymbol{x}'$, to a scalar (Murphy, 2012). The inputs can have any dimension: scalar, vector, or matrix. Typically, kernels provide similarity metrics between algebraic structures, e.g. cosine similarity for document comparisons, pyramid match kernels for bag-of-word image representations, and string kernels for strings (Murphy, 2012). In the GP context, kernels correspond to the similarity of functions of inputs, as opposed to the inputs themselves.

We define the *Gram matrix*, or covariance matrix, corresponding to data $\boldsymbol{X}$ and a kernel function

$k(x, x')$ as

$$\boldsymbol{K_{XX}} = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_N, x_1) & \cdots & k(x_N, x_N) \end{pmatrix}.$$

More generally, for any two sets of input data, $\boldsymbol{X}$ and $\boldsymbol{X'}$, we define $\boldsymbol{K_{XX'}}$ to be the kernel matrix where the (i, j) element is $k(\boldsymbol{x}_i, \boldsymbol{x}'_j)$.

By *Mercer's Theorem*, for any kernel k, there exists a function $\boldsymbol{\phi}$ mapping $\boldsymbol{x}$ to $\mathbb{R}^D$ such that $k(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x'})$ (Murphy, 2012); it is crucial to note that D may even be infinite dimensional. Thus, every kernel corresponds to a feature mapping of our inputs. When we use high-dimensional features for our data, it can be expensive to calculate our feature values at every dimension of a particular feature mapping. Thus, kernels may be more computationally efficient, as we evaluate a function as opposed to an inner product. When a kernel corresponds to an infinite-dimensional feature mapping, it actually enables us to compute something that would only be possible to approximate using straight-forward feature computation techniques.

Throughout this thesis, we will use the the *squared exponential* kernel, which has become one of the most popular kernels for GP regression (Yuan et al., 2009):

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \sigma_f^2 \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x'})^T \boldsymbol{M} (\boldsymbol{x} - \boldsymbol{x'})\right). \tag{2.3}$$

Here, σ_f^2 and $\boldsymbol{M}$ are referred to as the *hyper-parameters*, or kernel parameters. When $\boldsymbol{M}$ is a diagonal matrix, we denote the elements on the diagonal as l_n^{-2}, which we refer to as the characteristic length-scales of our GP (in Section 2.4, we discuss how length-scales control the smoothness of GP functions).

As an example, if our inputs are one-dimensional, the squared exponential is the following function:

$$k(x, x') = \sigma_f^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right). \tag{2.4}$$

To understand the corresponding feature space, take $\sigma_f^2 = 1$ and $l = 1$. Then, for x, x',

$$k(x, x') = e^{-\frac{x^2}{2}} e^{-\frac{x'^2}{2}} e^{xx'} = e^{-\frac{x^2}{2}} e^{\frac{x'^2}{2}} \sum_{k=0}^{\infty} \frac{(xx')^k}{k!}. \tag{2.5}$$

Thus, $k(x, x') = \boldsymbol{\phi}(x)^T \boldsymbol{\phi}(x')$ where

$$\boldsymbol{\phi}(x) = e^{-\frac{x^2}{2}} \left[1, x, \frac{x^2}{\sqrt{2}}, \frac{x^3}{\sqrt{6}}, \frac{x^4}{\sqrt{24}}, \dots\right]. \tag{2.6}$$

As we can see, the squared exponential covariance function maps an arbitrary input to an infinite-dimensional feature space.

2.4 Sampling from a Gaussian Process

Given a mean function $m(\boldsymbol{x})$ and a kernel $k(\boldsymbol{x}, \boldsymbol{x}')$, we can sample a function from any GP. To do this, we first define a set of inputs, $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, and evaluate the $N \times N$ covariance matrix, $\boldsymbol{K_{XX}}$, at all inputs. Defining $\boldsymbol{m_X}$ as the mean function evaluated at all inputs, we can then draw a random vector:

$$\boldsymbol{f} \sim \mathcal{N}(\boldsymbol{m_X}, \boldsymbol{K_{XX}})$$

where the n'th element of $\boldsymbol{f}$, f_n, corresponds to $f(\boldsymbol{x}_n)$. We can verify

$$E(f(\boldsymbol{x}_n)) = m(\boldsymbol{x}_n)$$

for all $n \in 1, \dots, N$, and

$$\text{Cov}(f(\boldsymbol{x}_n), f(\boldsymbol{x}_m)) = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$$

for all pairs $n, m \in 1, \dots, N$. By construction, every subset of $\boldsymbol{f}$ follows a multivariate normal distribution, so all the properties of a Gaussian process are satisfied. In this context, $\boldsymbol{f}$ is known as a *GP prior*, because we have not yet fit the GP to any data.

Figure 2.1 and Figure 2.2 represent draws from a GP prior with a constant (0) mean and a squared exponential kernel for various values of l and σ^2. The inputs $x_n \in \boldsymbol{X}$ are 1-dimensional, on the x-axis, and the corresponding outputs $f(x_n) \in \boldsymbol{f}$ are graphed on the y-axis.

Consider the behavior of the hyper-parameters. When l^2, which we refer to as the *length-scale*, decreases, the term in the exponent becomes more negative, meaning for similar input values, the function becomes less correlated. More specifically, the length-scale controls the distance in the input-space for which the function values become uncorrelated, so it essentially specifies the

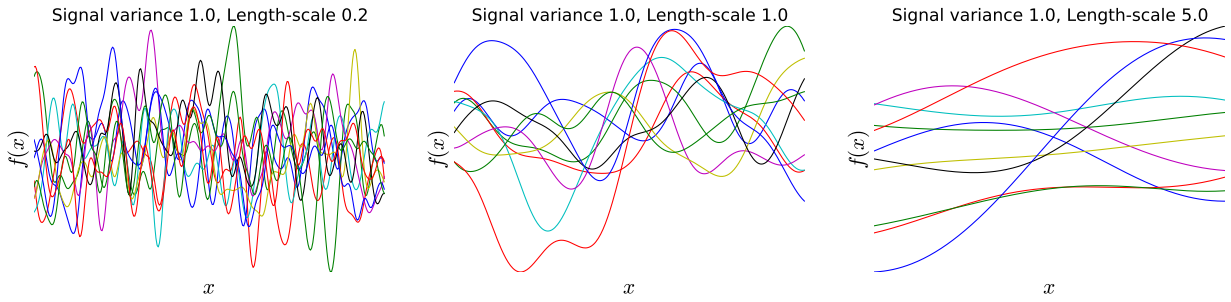

Figure 2.1: Random samples from GP priors using a squared exponential kernel with various length-scales. The length-scale controls the smoothness of our function – when it is small, points that are further away become less correlated, so the function is highly-varying. As the length-scale increases, the function becomes more smooth.

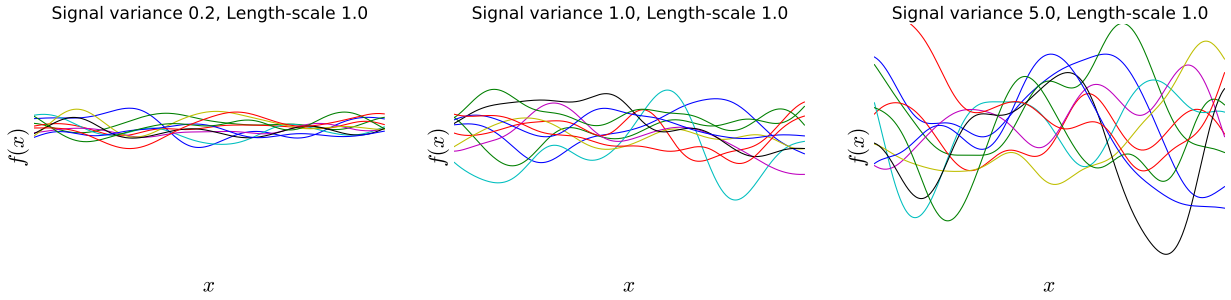

Figure 2.2: Random samples from GP priors using a squared exponential kernel with various signal variances. As the signal variance increases, the function values deviate further from the (constant) mean.

smoothness of the function (Rasmussen and Williams, 2006). Meanwhile, the *signal variance* σ_f^2 controls the variance of our function, so it determines how far we deviate from the mean (Rasmussen and Williams, 2006).

2.5 Gaussian Processes for Regression

Typically, we are not interested in drawing functions from a GP prior. Rather, we would like to estimate function values of a GP conditioned on some training data. In the simplest, noise-free case, we are given a set of inputs, $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, and corresponding function values $\boldsymbol{f} \in \mathbb{R}^N$. Assume a constant (0) mean function, and a covariance function $k(x, x')$ of our choice, where the set of kernel parameters is denoted as $\boldsymbol{\theta}$. We would like to estimate the function values $\boldsymbol{f}_*$ for a set of new inputs $\boldsymbol{X}_*$. Because finite subsets of GPs follow a multivariate normal distribution, we can

assume

$$\begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{0} \end{pmatrix}, \begin{pmatrix} \boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}} & \boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}_*} \\ \boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}} & \boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}_*} \end{pmatrix}\right).$$

Now, using the conditional properties of a multivariate normal, we know the conditional distribution of $\boldsymbol{f}_*$ is normally distributed, with:

$$P(\boldsymbol{f}_*|\boldsymbol{X}_*, \boldsymbol{X}, \boldsymbol{f}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}}\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}^{-1}\boldsymbol{f}, \boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}_*} - \boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}}\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}^{-1}\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}_*}). \tag{2.7}$$

We call this the *predictive distribution*, as we use it to model the distribution of an unobserved set of inputs, $\boldsymbol{X}_*$. With this information, we not only have an estimate of function values (by taking the mean of the conditional distribution), we have complete knowledge of the covariance structure. That is, the predictive distribution is in closed form, so values from this distribution can be easily sampled.

Typically, it may be more realistic that in our training data, we do not have access to the function values $f(\boldsymbol{x}_n)$ themselves, but rather noisy observations, $y_n = f(\boldsymbol{x}_n) + \epsilon_n$, where $\epsilon_n \sim \mathcal{N}(0, \sigma_n^2)$ i.i.d (Rasmussen and Williams, 2006). We can incorporate this noise into our model by adding σ_n^2 to every diagonal term in each covariance matrix, which corresponds to a kernel $k'(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}, \boldsymbol{x}') + \sigma_n^2 I(\boldsymbol{x} = \boldsymbol{x}')$, where $I(\cdot)$ is the indicator function. Once we have our updated kernel for regression with noise, the analysis is identical to Equation 2.7. Now, our observations are denoted as $\boldsymbol{y}$, and we are trying to predict values $\boldsymbol{y}_*$, where $\boldsymbol{K}$ corresponds to the new kernel:

$$P(\boldsymbol{y}_*|\boldsymbol{X}_*, \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}}\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}^{-1}\boldsymbol{y}, \boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}_*} - \boldsymbol{K}_{\boldsymbol{X}_*\boldsymbol{X}}\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}^{-1}\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}_*}). \tag{2.8}$$

2.6 Model Selection for Regression

An advantage of using Gaussian processes for regression is that we can compute the marginal likelihood of our model. Because we are assuming that, conditioning on our training data, $\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}})$, our marginal likelihood is

$$P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{(2\pi)^{N/2}|\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}|^{1/2}} \exp\left(-\frac{1}{2}\boldsymbol{y}^T\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}^{-1}\boldsymbol{y}\right). \tag{2.9}$$

We can tune our kernel parameters by setting them to the values that maximize the log marginal likelihood with respect to $\boldsymbol{\theta}$. This has clear advantages over non-probabilistic kernelized models like Support Vector Machines (SVMs), where it is not straightforward to compute the marginal likelihood in a probabilistic way. For these models, it can become expensive to tune hyper-parameters, as the most common way to do it is testing predictive accuracy over a grid of parameter values, which grows exponentially and tends to limit the number of hyper-parameters to no more than 2 or 3 in practice (Hsu et al., 2003). Meanwhile, by maximizing the marginal likelihood used for GP regression, we can feasibly include more hyper-parameters, albeit with a slow training time (Rasmussen and Williams, 2006). Thus, being able to calculate the marginal likelihood is a crucial characteristic for GPs, as it can help significantly with model selection.

An additional advantage is that by calculating the marginal likelihood, we implicitly integrate over the function values at the unobserved function locations (Duvenaud, 2014). That is, we are implicitly making the following calculation:

$$P(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta}) = \int P(\boldsymbol{y}|\boldsymbol{f})P(\boldsymbol{f}|\boldsymbol{X},\boldsymbol{\theta})d\boldsymbol{f} = \mathcal{N}(\boldsymbol{y}|\boldsymbol{0},\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}). \tag{2.10}$$

Therefore, we are considering the entire range of function values (our hypotheses) when we evaluate the marginal likelihood. As a result, our marginal likelihood includes a tradeoff between fit and model complexity. Taking the log marginal likelihood, we see

$$\log P(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta}) = -\frac{N}{2}\log 2\pi - \frac{1}{2}\log|\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}| - \frac{1}{2}\boldsymbol{y}^T\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}}^{-1}\boldsymbol{y}. \tag{2.11}$$

The first term is a normalizing constant. The second term penalizes model complexity, as more complicated covariance structures will yield larger determinants. The final term is the only term that involves the output data, and it controls the quality of our fit (Rasmussen and Williams, 2006).

Being able to compute the marginal likelihood is a general advantage of nonparametric Bayesian techniques, and it allows us to incorporate uncertainty into our model. Therefore, overfitting tends to be less significant a problem in GP regression, because a relatively small number of parameters needs to be estimated (Duvenaud, 2014).

In Figure 2.3, we fit a GP on toy sigmoidal data. We use the squared exponential kernel,

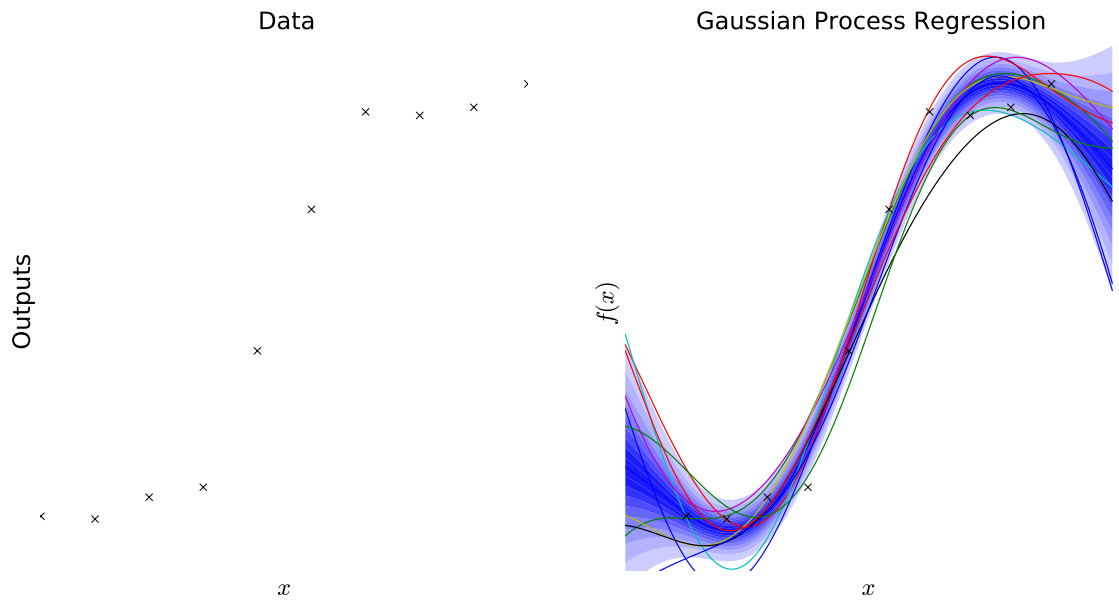

Figure 2.3: On the left, data from a sigmoidal curve with noise. On the right, samples from a GP trained on the data (represented by 'x'), using a squared exponential covariance function. The shaded regions correspond to deciles of the predictive distribution, and the lines represent samples drawn from the predictive distribution. After optimizing the marginal likelihood, we learn kernel parameters $\sigma_n^2 = .014$, $\sigma_f^2 = .323$, *and* $l^2 = 21.6$.

learning the length-scale and noise-scale by optimizing the marginal likelihood.

Chapter 3

Deep Gaussian Processes

In recent years, *deep learning* methods have become popular in supervised learning settings, largely due to successful applications in vision, text, and speech (Krizhevsky et al., 2012; Mikolov et al., 2010; Hinton et al., 2012). In this chapter, we introduce *deep Gaussian processes*, which possess deep architectures yet retain many of the advantages of Gaussian processes. By composing GP units with one another, a deep GP is able to learn more complex functions, specifically where data is non-stationary. We then compare deep GPs to neural networks and highlight the motivations for and benefits of models that possess a deep architecture, along with their limitations.

3.1 Definition

A deep Gaussian process is, intuitively, a distribution composed of a cascade of GPs, divided into layers in a network architecture. For an L layer deep GP, the distribution for a single output, y_n, for a single data point, $\boldsymbol{x}_n \in \mathbb{R}^{D^{(0)}}$, consists of three types of layers. There is one *input layer*, $\boldsymbol{x}_n$. There are $L-1$ hidden layers, $\{\boldsymbol{h}_n^l\}_{l=1}^L$, each of which has a specified number of *hidden units*, which correspond to the dimensionality of the layer. Finally, there is an output layer, y_n, which is connected to the last hidden layer. Each layer is completely connected by GPs, each with their own kernel.

As a concrete example, consider the simplest type of deep GP, with a one dimensional input, x_n, a one dimensional hidden unit, h_n, and a one dimensional output, y_n. This two-layer network

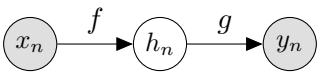

Figure 3.1: A graphical representation of a simple two-layer deep GP where every layer has one dimension. The arrows represent the directions of the functions. x_n and h_n are connected by the function $f \sim \mathcal{GP}(0, k^{(1)}(x, x'))$, while h_n and y_n are connected by $g \sim \mathcal{GP}(0, k^{(2)}(h, h'))$. Therefore, $y_n = g(f(x_n))$.

consists of two GPs, f and g. More specifically,

$$h_n = f(x_n), \text{ where } f \sim \mathcal{GP}(0, k^{(1)}(x, x'))$$

and

$$y_n = g(h_n), \text{ where } g \sim \mathcal{GP}(0, k^{(2)}(h, h')).$$

Because each GP is allowed to have different hyper-parameters, note that the covariance matrices correspond to different kernels. The graphical representation of this model is depicted in Figure 3.1.

More concretely, we can define a deep GP as the distribution over functions constructed by composing GPs. Throughout this thesis, we will assume that each function is drawn independently, and that data is generated by the following process (Duvenaud, 2014):

$$\boldsymbol{f}^{(1:L)}(\boldsymbol{x}) = \boldsymbol{f}^{(L)}(\boldsymbol{f}^{(L-1)}(\dots \boldsymbol{f}^{(2)}(\boldsymbol{f}^{(1)}(\boldsymbol{x}))\dots))$$

$$\text{where } f_d^{(l)} \sim \mathcal{GP}\left(0, k_d^{(l)}(\boldsymbol{x}, \boldsymbol{x}')\right) \text{ for } f_d^{(l)} \in \boldsymbol{f}^{(l)}.$$

Unpacking the notation, in addition to our input $\boldsymbol{x}$ and output $\boldsymbol{y}$, there are $L-1$ hidden layers. Each layer l consists of $D^{(l)}$ GPs, where $D^{(l)}$ is the number of units at layer l. The notation $\boldsymbol{f}^{(l)}(\boldsymbol{x})$ refers to evaluating all the GPs at layer l, storing the result as a vector of length $D^{(l)}$. Each layer has an input dimension and an output dimension; the input dimension of the input layer is the dimensionality of $\boldsymbol{x}$, $D^{(0)}$, while we are assuming the output layer has dimension 1. Otherwise, the input dimension, also known as the number of units, of layer l is the output dimension of layer $l-1$, while the output dimension of layer l is the number of functions in $\boldsymbol{f}^{(l)}$. Thus, each function $f_d^{(l)} \in \boldsymbol{f}^{(l)}$ takes as input a vector whose dimension is the output dimension of layer $l-1$, and

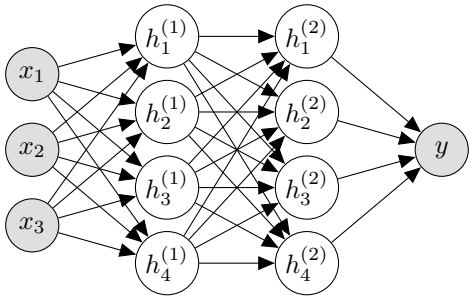

Figure 3.2: Graphical representation of a more complicated deep GP architecture. As in Figure 3.1, every edge corresponds to a GP between units, as the outputs of each layer are the inputs of the following layer. Our input data is 3-dimensional, while the two hidden layers in this model each have 4 hidden units. The layers are completely connected.

returns a scalar. We note that every GP has its own specified kernel and kernel parameters, so there can be multiple kernels per layer.

To draw samples from a deep GP, we start with our input, $\boldsymbol{x}$. For every layer, we sample from the previous layer in a similar fashion. For unit d in layer l, we draw a sample from $f_d^{(l)}$, using the concatenation of function values in layer $l-1$ as our input vector. We do the sampling by drawing a Gaussian with the specified mean and covariance. We repeat this process until the output layer, which leaves us with a scalar output. Figure 3.3 depicts values drawn from the two-layer deep GP depicted in Figure 3.1.

One advantage of using a deep GP prior over a standard GP is that deep GPs can model non-stationary functions without the use of a non-stationary kernel. Intuitively, a stationary kernel is one that is invariant to input transformations; that is, it only compares the relative locations of inputs $\boldsymbol{x}$ and $\boldsymbol{x}'$ (Rasmussen and Williams, 2006). If the shape of a function changes along the input space, a standard GP would require a non-stationary kernel to capture these differences. However, because a deep GP is the composition of GPs with different kernels, it can model non-stationary functions. Since the outputs of each layer are used as inputs to the next, and can thus be transformed in different ways by various kernel parameters, the function is no longer constrained to take on one particular shape.

To train a deep GP for regression, we would like to integrate out the hidden function values at each layer, because they are unknown. We would then like to use the conditional distribution to evaluate the predictive mean and variance for a new data point. However, evaluating the marginal likelihood is intractable, creating a significant obstacle for fitting. We discuss fitting mechanisms

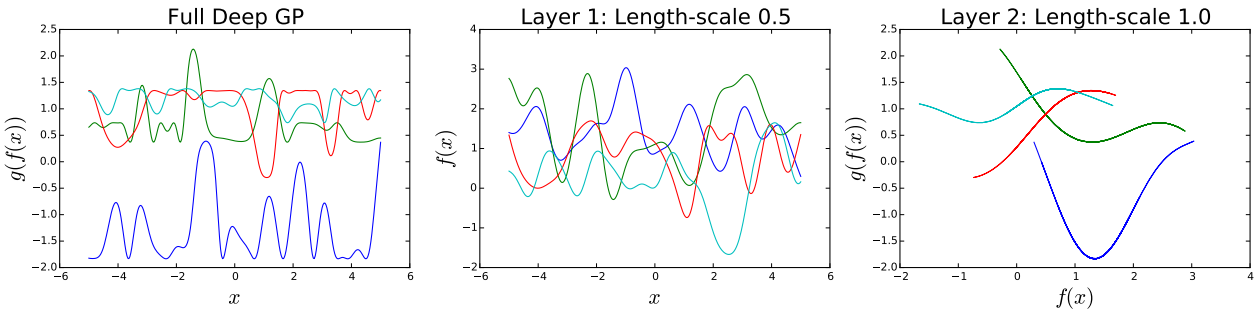

Figure 3.3: 4 functions sampled from the single-dimension two-layer deep GP architecture from Figure 3.1. The graph on the furthest left represents the final draws, which correspond to the composition of f and g, displayed center and right, respectively. The colors of each function draw are consistent throughout. As we can see, the function draws are non-stationary – their shape and curvature varies as a function of the input space. The squared exponential kernel is used for both GPs, with signal variance 1.

for deep GPs for regression, along with approximations to circumvent the intractability problem, in Chapter 4.

3.2 Neural Networks

The deep architecture of a deep GP mirrors that of a *neural network*. The simplest neural network is known as the perceptron, which has the following form for an input vector $\boldsymbol{x} \in \mathbb{R}^{d_{in}}$:

$$f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}. \tag{3.1}$$

Here, $\boldsymbol{W} \in \mathbb{R}^{d_{out} \times d_{in}}$ is known as the weight matrix, and $\boldsymbol{b} \in \mathbb{R}^{d_{out}}$ is known as the bias vector (Goldberg, 2015). As we can see, this is a linear function, and the setup is very similar to that of linear regression. To learn more complex functions, we introduce nonlinearities, applying a sigmoid function $\sigma : \mathbb{R} \to \mathbb{R}$ element wise:

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}). \tag{3.2}$$

Neural networks extend the perceptron by introducing multiple layers, where the inputs for one layer are the outputs of the preceding layer. Consider the following one-layer feed-forward neural

network:

$$f(\boldsymbol{x}) = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2 \tag{3.3}$$

$$\boldsymbol{h} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1). \tag{3.4}$$

Above, $\boldsymbol{h}$ is known as the *hidden layer*, with dimension d_{hidden}. Following the notation of Goldberg (2015), for $\boldsymbol{x} \in \mathbb{R}^{d_{in}}$ and $f(\boldsymbol{x}) \in \mathbb{R}^{d_{out}}$, we have $\boldsymbol{W}_1 \in \mathbb{R}^{d_{hidden} \times d_{in}}, \boldsymbol{W}_2 \in \mathbb{R}^{d_{out} \times d_{hidden}}$, $\boldsymbol{b}_1 \in \mathbb{R}^{d_{hidden}}$, $\boldsymbol{b}_2 \in \mathbb{R}^{d_{out}}$. In the 2-layer model, the role of $\sigma(\cdot)$ is clear. Without a nonlinear term, our function would be the composition of two linear functions, which is in itself linear. By introducing a nonlinearity, we can extend the number of layers and learn more complicated functions.

Extending this idea to multiple layers, we can define hidden layer $\boldsymbol{h}^{(l)}$ as

$$\boldsymbol{h}^{(l)}(\boldsymbol{x}) = \sigma(\boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)}(\boldsymbol{x}) + \boldsymbol{b}^{(l)}) \tag{3.5}$$

where $\boldsymbol{h}^{(0)}(\boldsymbol{x}) = \boldsymbol{x}$ (Duvenaud, 2014). The output of this network is thus the output of the last layer. The parameters we are trying to learn are the sets of unknown matrices $\boldsymbol{W}$ and biases $\boldsymbol{b}$, and they are typically learned through the *back-propagation* algorithm. The input dimension of any particular layer is also known as the number of *hidden units*.

What is the advantage of a deep architecture? If we are trying to learn a function that is complex and highly-varying, a linear approximation could require a large number of piece-wise functions (Bengio and LeCun, 2009). Additionally, Bengio and LeCun (2009) show that it is possible to fit highly-varying functions via the composition of non-linear functions, requiring fewer parameters than a one-layer linear approximation. This composition allows for non-local learning, meaning we can generalize and learn the function value for inputs that are not close to our training examples. Thus, not only do deep architectures require fewer parameters to learn complex functions, but they also require less training examples.

3.3 Comparing Deep Models

Deep GPs and neural networks share many common qualities. The most obvious is in their deep architecture – both models consist of the composition of single-layer functions, where the outputs

for one layer are used as the inputs for the forthcoming layer. Additionally, they can both feature architectures where each layer is completely connected to the layer before it.

In fact, it can be shown that certain neural network architectures correspond to Gaussian processes, when the weights in the neural network $\boldsymbol{w}$ are unknown and random (Neal, 2012). Consider a neural network with one hidden layer and K hidden units. We are assuming the set of weights are i.i.d., with zero mean and finite variance σ^2, and that the features $\boldsymbol{h}(\boldsymbol{x}) = [h_1(\boldsymbol{x}), \ldots, h_K(\boldsymbol{x})]^T$ are fixed. Consider the following model:

$$f(\boldsymbol{x}) = \frac{1}{K}\boldsymbol{w}^T\boldsymbol{h}(\boldsymbol{x}) = \frac{1}{K}\sum_{k=1}^{K} w_k h_k(\boldsymbol{x}). \tag{3.6}$$

We would like to calculate the distribution over the function values of two inputs, $\boldsymbol{x}$ and $\boldsymbol{x}'$. We first note that by the Central Limit Theorem, as K grows, $f(\boldsymbol{x})$ approaches a normal distribution:

$$f(\boldsymbol{x}) \to \mathcal{N}\left(0, \frac{\sigma^2}{K}\sum_{k=1}^{K} h_k(\boldsymbol{x})^2\right). \tag{3.7}$$

Because the weights are i.i.d, we can extend this logic to the two-dimensional multivariate normal (Neal, 2012):

$$\begin{pmatrix} f(\boldsymbol{x}) \\ f(\boldsymbol{x}') \end{pmatrix} \to \mathcal{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \frac{\sigma^2}{K}\begin{pmatrix} \sum_{k=1}^{K} h_k(\boldsymbol{x})h_k(\boldsymbol{x}) & \sum_{k=1}^{K} h_k(\boldsymbol{x})h_k(\boldsymbol{x}') \\ \sum_{k=1}^{K} h_k(\boldsymbol{x}')h_k(\boldsymbol{x}) & \sum_{k=1}^{K} h_k(\boldsymbol{x}')h_k(\boldsymbol{x}') \end{pmatrix}\right). \tag{3.8}$$

We have established that any set of function values in this neural network architecture follow a multivariate normal as $K \to \infty$. This is exactly the definition of a Gaussian process. Thus, this neural network architecture is equivalent to a GP as K becomes large. Because a single-layer neural network is a universal approximation – that is, it can approximate continuous functions on subsets of $\mathbb{R}^n$ (Cybenko, 1989) – a GP has similar universal approximation qualities. A deep GP can thus be viewed as a neural network where every other layer contains an infinite number of hidden units, with the same weight conditions we posed above (Duvenaud, 2014).

Therefore, a deep GP uses a deep architecture to retain the benefits of a deep neural network, with the hope that fewer training examples can be used to learn complicated functions (Damianou and Lawrence, 2013). There are many concrete differences between neural networks and deep GPs.

Most notably, a deep GP is a nonparametric model, meaning the data generation process does not rely on a rich set of parameters like it does for a neural network. Rather, because we are trying to model a continuous function, the parameter space can be considered infinite. Additionally, while a neural network has to introduce a nonlinear term to learn nonlinear functions, nonlinearities arise in deep GPs automatically via the composition of random functions. Meanwhile, deep GPs require the user to specify a kernel for every function, whereas no comparable task exists for neural networks.

However, perhaps the biggest differences arise from training. Fitting a deep neural network can become complicated for numerous reasons: we are required to learn a weight matrix and bias terms for every layer, so the number of parameters can become quite large as the size of the input-dimension and number of layers increases. This requires an immense amount of computing power and time, both to train and test for large datasets (Livni et al., 2014). While the parametric nature of neural networks can be advantageous for large data sets because the size of the parameter space is constant, overfitting can become an issue since we are required to estimate a large number of parameters (Krizhevsky et al., 2012). As a result, fitting a deep neural network is sometimes referred to as "black magic," because it can be difficult to know the number of layers, dimensionality of each layer, and the choice of nonlinearities that will give rise to the best fit (Simard et al., 2003).

Meanwhile, training a deep GP is intractable if we would like to integrate out the hidden function values (Damianou and Lawrence, 2013). However, as we will see, approximation techniques can make training feasible. A deep GP retains many of the structural advantages of a single-layer GP. For example, overfitting is less of an issue if we can optimize the marginal likelihood with respect to the kernel parameters. Additionally, Damianou and Lawrence (2013) show that the Bayesian nature of deep GPs can provide a better fit when data is scarce. Finally, because we can evaluate the marginal likelihood of the model, we can train to not only find the optimal set of kernel parameters, but we can additionally evaluate the optimal number of layers in the architecture along with the number of functions per layer.

Chapter 4

Implementation

In the previous two chapters, we have gone over the background and theory behind Gaussian processes and deep Gaussian processes for regression. In this chapter, we discuss deep GP implementation, along with the two main challenges posed by inference: the intractability of integrating out the hidden function values, and the expensive computational complexity of fitting a deep architecture. First, we detail the implementation of a standard GP, focusing on why the problem is tractable. Next, we discuss the Fully Independent Training Conditional (FITC) method, which provides an approximation to make fitting a GP more feasible. Then, we introduce the Deep Gaussian Process Sampling algorithm (DGPS) used to fit a deep GP, which uses Monte Carlo sampling to overcome the intractability hurdle; rather than treating the hidden values as parameters that should be learned, we use sampling to preserve the Bayesian nature of the model. Additionally, we use the FITC approximation to improve the computational complexity, putting a prior on the pseudo data to improve the approximation. We then provide pseudocode for general architectures, after which we walk through the specific case of fitting a 2-layer deep GP on one dimensional data.

4.1 Implementing a Gaussian Process

To implement a standard Gaussian process for regression, we first optimize the marginal likelihood with respect to the kernel parameters $\boldsymbol{\theta}$, after which we compute the conditional distribution to learn the predictive mean and predictive variance of a new input. Recall, the marginal likelihood

is given by

$$P(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta}) = \int P(\boldsymbol{y}|\boldsymbol{f})P(\boldsymbol{f}|\boldsymbol{X},\boldsymbol{\theta})d\boldsymbol{f} \tag{4.1}$$

$$= \frac{1}{(2\pi)^{N/2}|\boldsymbol{K_{XX}}|^{1/2}} \exp\left(-\frac{1}{2}\boldsymbol{y}^T\boldsymbol{K_{XX}^{-1}}\boldsymbol{y}\right). \tag{4.2}$$

In the standard GP, we can successfully integrate out the function values $\boldsymbol{f}$, so the marginal likelihood is possible to compute analytically. We will see later that deep GPs do not share this quality when trying to integrate out the hidden values. Now, to optimize with respect to $\boldsymbol{\theta}$, we can instead optimize the log marginal likelihood, because the logarithm function is monotone increasing:

$$\log P(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta}) = -\frac{N}{2}\log(2\pi) - \frac{1}{2}\log\left(|\boldsymbol{K_{XX}}|\right) - \frac{1}{2}\boldsymbol{y}^T\boldsymbol{K_{XX}^{-1}}\boldsymbol{y}. \tag{4.3}$$

Taking the partial derivative with respect to kernel parameter θ_j, we obtain (Rasmussen and Williams, 2006):

$$\frac{\partial}{\partial\theta_j}\log P(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta}) = \frac{1}{2}\mathrm{tr}\left(\left(\boldsymbol{\alpha}\boldsymbol{\alpha}^T - \boldsymbol{K_{XX}^{-1}}\right)\frac{\partial\boldsymbol{K_{XX}}}{\partial\theta_j}\right) \text{ where } \boldsymbol{\alpha} = \boldsymbol{K_{XX}^{-1}}\boldsymbol{y}. \tag{4.4}$$

We can thus optimize the log marginal likelihood using a numerical optimization method such as conjugate gradients, which is applicable because the matrix $\boldsymbol{K_{XX}}$ is symmetric positive definite (Shen et al., 2006).

We note that any optimization method requires a matrix inversion to compute the partial derivatives, which takes time $\mathcal{O}(N^3)$ for N the number of points in the training set (Rasmussen and Williams, 2006). On top of that, it may take many epochs for the optimization to converge. Once we have inverted $K_{\boldsymbol{XX}}$, to predict the mean and variance of a new data point, we only compute dot products between matrices and vectors, which takes time $\mathcal{O}(N)$ for every new point (Snelson and Ghahramani, 2005).

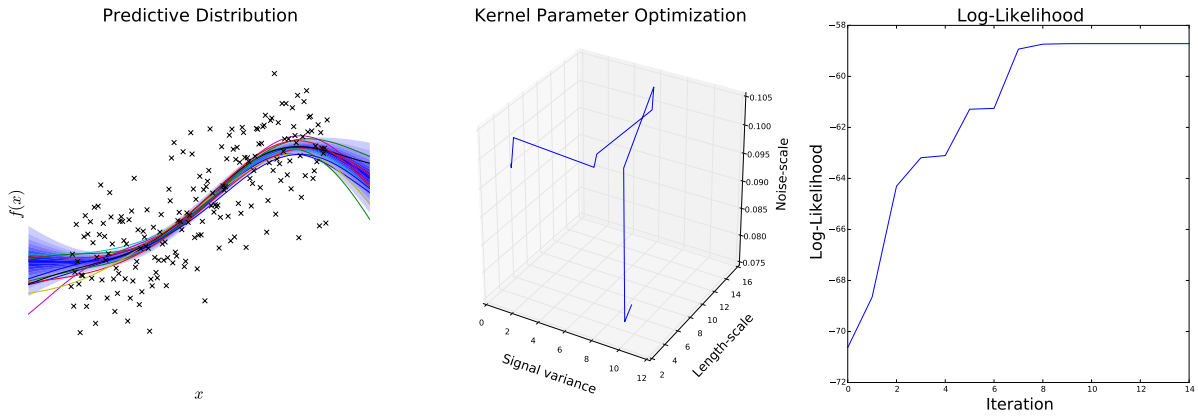

Figure 4.1: Fitting a 1-dimensional GP for regression on sigmoidal data using the squared exponential kernel. The plot on the far left shows the data along with samples and deciles from the predictive distribution. The plot in the middle displays the values of the kernel parameters throughout the optimization, which uses conjugate gradients. The plot on the right shows the log-likelihood of the data $P(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$ as a function of time.

4.2 Fully Independent Training Conditional Approximation

The $\mathcal{O}(N^3)$ training time required to invert the covariance matrix in GP regression leaves much to be desired. GP regression is significantly slower than many other regression techniques, so it is not frequently used for datasets with more than a few thousand training examples (Quinonero-Candela et al., 2007). This is one of the fundamental disadvantages of using GPs for regression – the standard algorithm is infeasible for most modern problems involving large datasets.

We instead rely on approximation techniques to fit a GP regression model. While some solutions involve using a subset of the data, either at random or through a greedy technique, we focus on a technique involving *pseudo data*, where our training samples may not be in the original data. We call it the Fully Independent Training Conditional (FITC) approximation (Quinonero-Candela et al., 2007). Rather than learning the model for our N training inputs and corresponding outputs, we learn it for a pseudo data set of size M. We introduce M *pseudo inputs* $\bar{\boldsymbol{X}} = \{\bar{\boldsymbol{x}}_m\}_{m=1}^M$ and the corresponding *pseudo outputs* $\bar{\boldsymbol{y}} = \{\bar{y}_m\}_{m=1}^M$, which correspond to the function values at the pseudo inputs (Snelson and Ghahramani, 2005).

We assume that, conditioning on the pseudo inputs and outputs, every observed output value y_n is generated from the corresponding input $\boldsymbol{x}_n$ by conditioning a multivariate normal on the

pseudo data:

$$P(y_n|\boldsymbol{x}_n, \bar{\boldsymbol{X}}, \bar{\boldsymbol{y}}) = \mathcal{N}(\boldsymbol{k}_{\boldsymbol{x}_n}^T \boldsymbol{K}_{\bar{\boldsymbol{x}}\bar{\boldsymbol{x}}}^{-1} \bar{\boldsymbol{y}}, k_{\boldsymbol{x}_n} - \boldsymbol{k}_{\boldsymbol{x}_n}^T \boldsymbol{K}_{\bar{\boldsymbol{x}}_n\bar{\boldsymbol{x}}_n}^{-1} \boldsymbol{k}_{\boldsymbol{x}_n} + \sigma^2) \tag{4.5}$$

for $k_{\boldsymbol{x}_n} = k(\boldsymbol{x}_n, \boldsymbol{x}_n), \boldsymbol{k}_{\boldsymbol{x}_n} = [k(\boldsymbol{x}_n, \bar{\boldsymbol{x}}_1), \ldots, k(\boldsymbol{x}_n, \bar{\boldsymbol{x}}_m)]$, and $\boldsymbol{K}_{\bar{\boldsymbol{x}}\bar{\boldsymbol{x}}}$ the kernel matrix evaluated at our pseudo inputs (Snelson and Ghahramani, 2005). Now, the fundamental assumption is that, given the pseudo data, the training data is generated *independently*:

$$P(\boldsymbol{y}|\boldsymbol{X}, \bar{\boldsymbol{X}}, \bar{\boldsymbol{y}}) = \prod_{n=1}^{N} P(y_n|\boldsymbol{x}_n, \bar{\boldsymbol{X}}, \bar{\boldsymbol{y}}). \tag{4.6}$$

We assume the data is generated independently because now the covariance matrix of our outputs is diagonal. This is a boon to the computational complexity, as it removes the $\mathcal{O}(N^3)$ matrix inversion.

In the method posed by Snelson and Ghahramani (2005), they learn the hyper-parameters and the pseudo inputs, but integrate out the pseudo outputs when obtaining the predictive mean and variance for a new data point. They do this by putting a normal prior on the pseudo outputs. Specifically, they let

$$P(\bar{\boldsymbol{y}}|\bar{\boldsymbol{X}}) = \mathcal{N}\left(\boldsymbol{0}, \boldsymbol{K}_{\bar{\boldsymbol{X}}\bar{\boldsymbol{X}}}\right). \tag{4.7}$$

Using the kernel matrix evaluated at the pseudo inputs as a covariance matrix encourages the shape of the pseudo data to resemble that of the real data. This has some regularization benefits as well; without the prior, we would risk the outputs spreading and not resembling our input data, resulting in overfitting.

Thus, this model requires the user to learn two types of parameters: the kernel parameters, $\boldsymbol{\theta}$, as before, and the location of the pseudo inputs, $\bar{\boldsymbol{X}}$. We find these values by optimizing the marginal likelihood using a gradient descent method. Without going into the details for integrating out the pseudo outputs, computing the predictive mean and variance for the value of a new point y_* at input $\boldsymbol{x}_*$ no longer requires the inversion of a $N \times N$ positive-definite matrix (Snelson and Ghahramani, 2005). Rather, the computational cost is dominated by inverting a matrix of size M, of our pseudo inputs, which is diagonal by our independence assumption. As a result, training requires time $\mathcal{O}(NM^2)$ and prediction requires time $\mathcal{O}(M^2)$ (Snelson and Ghahramani, 2005).

In Figure 4.2, we use the method described above to plot the predictive mean of a GP trained

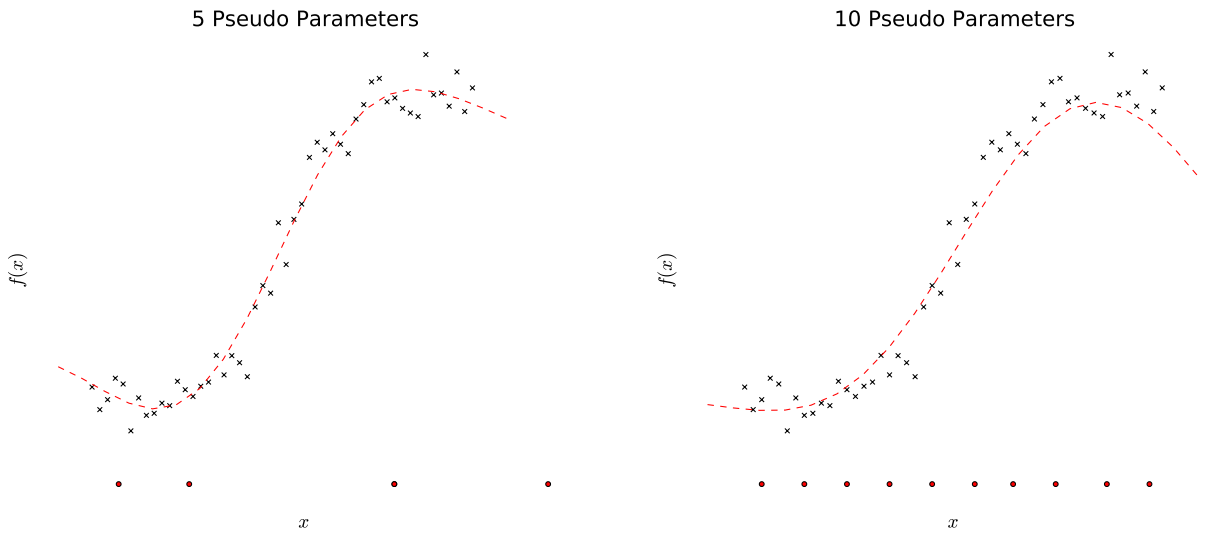

Figure 4.2: The predictive mean of a GP trained on sigmoidal data ($N = 50$) using the FITC approximation. On the left, we use 5 pseudo data points, while on the right, we use 10. The data is denoted by 'x', and the pseudo data inputs are the red circles beneath. Note that we integrate out over the pseudo outputs, and we learn the pseudo inputs by maximum likelihood, so only their input locations (on the x-axis) are plotted. In both plots, the pseudo inputs spread out to cover the input space.

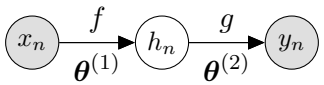

Figure 4.3: A two-layer deep GP with a single hidden dimension.

using the FITC approximation, varying the number of pseudo data points.

4.3 Deep Gaussian Processing Sampling Algorithm

4.3.1 Motivation

Thus, it is straightforward to implement a single-layer GP for regression, using the FITC approximation to improve the computational complexity. Unfortunately, exact inference on a deep GP is not possible, because it is intractable to integrate out the hidden function values at every hidden layer.

Consider a deep GP with a single hidden layer and 1-dimensional inputs, as in Figure 4.3. That is, $\boldsymbol{X} = \{x_n\}_{n=1}^N$ and $\boldsymbol{y} = \{y_n\}_{n=1}^N$. Every pair of inputs and outputs contains a corresponding hidden value $h_n \in \boldsymbol{H}$ (note that $\boldsymbol{H}$ is $N \times 1$). We denote as f the zero-mean GP that connects $\boldsymbol{X}$

and $\boldsymbol{H}$ (with kernel parameters $\boldsymbol{\theta}^{(1)}$), and we denote as g the zero-mean GP that connects $\boldsymbol{H}$ and $\boldsymbol{y}$ (with kernel parameters $\boldsymbol{\theta}^{(2)}$).

Ideally, a Bayesian treatment would allow us to integrate out the hidden function values to evaluate $P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})$. Using $\boldsymbol{\theta} = (\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)})$:

$$P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = \int P(\boldsymbol{y}|\boldsymbol{H}, \boldsymbol{\theta}^{(2)}) P\left(\boldsymbol{H}|\boldsymbol{X}, \boldsymbol{\theta}^{(1)}\right) d\boldsymbol{H} \tag{4.8}$$

$$= \int \mathcal{N}\left(\mathbf{0}, \boldsymbol{K_{HH}}\right) \mathcal{N}\left(\mathbf{0}, \boldsymbol{K_{XX}}\right) d\boldsymbol{H}. \tag{4.9}$$

Unfortunately, this computation, which involves the integrals of Gaussians with respect to kernel functions, is intractable (Damianou and Lawrence, 2013). As a result, we cannot use this fully Bayesian treatment to fit the data.

To circumvent these challenges, we introduce the Deep Gaussian Process Sampling algorithm (DGPS), which relies on two central ideas.

First, because we cannot integrate out hidden function values, we rely on sampling predictive means and covariances to approximate the marginal likelihood. Using these samples, we use automatic differentiation techniques to evaluate the gradients and optimize our objective. This algorithm is straightforward, and it mirrors the data generating process of a deep GP: we sample values until we reach the final layer.

Second, if we fit every GP in our model without any approximations, the computational complexity for L layers and H hidden units per layer is $\mathcal{O}(N^3LH)$, which becomes infeasible quickly as the size of the training data, the number of layers, or the number of hidden units increases. To ease the computational burden of our model, we replace every GP with the FITC GP described in Section 4.2. Although we are now required to learn M pseudo inputs and outputs for every GP, it is a more computationally feasible solution, with complexity $\mathcal{O}(N^2MLH)$ where $M << N$.

4.3.2 Related Work

While Gaussian processes are a well-studied model for regression, deep Gaussian processes are much newer, having been introduced in 2013 by Damianou and Lawrence (2013), who use the FITC approximation at every layer to fit a deep GP. The DPGS algorithm shares this quality. Rather than learning all the pseudo outputs, Damianou and Lawrence (2013) perform inference

using approximate variational marginalization. Thus, they put a distribution over their latent parameters, learning hyper-parameters at every pseudo data point. As a result, they integrate out the pseudo outputs, a property the DGPS algorithm does not share.

Subsequent methods implemented to train deep GPs for regression (Hensman and Lawrence, 2014; Dai et al., 2015; Bui et al., 2016) also use variational approximations. The differences between these methods relate to the inference method used. Hensman and Lawrence (2014) propose a nested variational compression, while Dai et al. (2015) and Bui et al. (2016) use an auto-encoder and expectation propagation, respectively.

While the variational methods benefit from being able to integrate out the pseudo outputs, they rely on integral approximations that depend on the particular kernel being used. Thus, they do not extend easily to arbitrary kernels or to those that are hard to integrate, such as the Matérn kernel. Meanwhile, the DGPS algorithm uses Monte Carlo sampling instead of a variational approximation. For one, sampling is a simpler technique to implement and more intuitive to understand. Additionally, the DGPS algorithm can extend easily to most kernels. Because the DGPS algorithm relies on automatic differentiation as opposed to automatic integration (which is a much harder problem, if not impossible), it is better suited for a more diverse set of kernel functions.

4.3.3 Sampling Hidden Values

As the name suggests, the Deep Gaussian Process Sampling algorithm uses sampling as a framework both to approximate the model marginal likelihood and to evaluate gradients with respect to current parameter values.

Specifically, consider an arbitrary architecture with L layers. We would like to evaluate the marginal likelihood of our training data in the following way: For inputs $\boldsymbol{X}$, we calculate the predictive mean and predictive covariance for every unit in the first hidden layer. Then, we sample values from each unit's predictive distribution, and use these as inputs to the following layer. We repeat this process until we have a single predictive mean $\tilde{\boldsymbol{\mu}}$ and predictive covariance $\tilde{\boldsymbol{\Sigma}}$ for the last layer, corresponding to our output. Then, we can evaluate

$$P(\boldsymbol{y}|\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}}) = \mathcal{N}(\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}}). \tag{4.10}$$

This procedure will generate a single sample used to evaluate the model marginal likelihood. To get a more accurate approximation, we use K different samples, $\{(\tilde{\boldsymbol{\mu}}_k, \tilde{\boldsymbol{\Sigma}}_k)\}_{k=1}^K$ and approximate the marginal likelihood by taking the average marginal likelihood with respect to these samples

$$P(\boldsymbol{y}|\boldsymbol{X}) \approx \sum_{k=1}^{K} P(\boldsymbol{y}|\tilde{\boldsymbol{\mu}}_{\boldsymbol{k}}, \tilde{\boldsymbol{\Sigma}}_{\boldsymbol{k}}). \tag{4.11}$$

Finally, we can take gradients of the above term with respect to our model parameters, which consist of kernel parameters for every GP along with pseudo inputs and outputs (described in Section 4.3.4). It may not appear possible to take the derivative of a random sample with respect to one of the parameters that it is being sampled from. For example, if $x \sim \mathcal{N}(\mu, \sigma^2)$, how can we evaluate $\partial x/\partial\sigma$? We note that we can also represent x in terms of a standard normal sample $z \sim \mathcal{N}(0, 1)$, so

$$x = \sigma z + \mu \Rightarrow \frac{\partial x}{\partial \sigma} = \frac{\partial}{\partial \sigma}(\sigma z + \mu) = z. \tag{4.12}$$

Using this trick, not only can we approximate the marginal likelihood of our deep model, but we can also take the gradient with respect to current parameters.

4.3.4 FITC for Deep GPs

To make this fitting mechanism more scalable, we replace every GP in our model with a FITC GP. Now, for each GP, corresponding to hidden unit d in layer l, we introduce pseudo inputs $\bar{\boldsymbol{X}}_d^{(l)}$ and corresponding pseudo outputs $\bar{\boldsymbol{y}}_d^{(l)}$. Additionally, we put a normal prior on the location of the pseudo outputs, as in Equation 4.7. Now, the prior over the entire set of pseudo outputs conditioning on the pseudo inputs is simply the product of each individual prior:

$$P\left(\{\bar{\boldsymbol{y}}_d^{(1)}\}_{d=1}^{D^{(1)}}, \ldots, \{\bar{\boldsymbol{y}}_d^{(L)}\}_{d=1}^{D^{(L)}} | \{\bar{\boldsymbol{X}}_d^{(1)}\}_{d=1}^{D^{(1)}}, \ldots, \{\bar{\boldsymbol{X}}_d^{(L)}\}_{d=1}^{D^{(L)}}\right) = \prod_{l=1}^{L}\prod_{d=1}^{D^{(l)}} P\left(\bar{\boldsymbol{y}}_d^{(l)} | \bar{\boldsymbol{X}}_d^{(l)}\right) \tag{4.13}$$

Ideally, we would like to follow Snelson and Ghahramani (2005) and integrate over the pseudo outputs. However, in the multiple-layer case, this integral must be approximated (Damianou and Lawrence, 2013), as it is not as straightforward computationally as for a single-layer GP . We can approximate the integral with respect to specific kernels, but this technique would then not extend

to arbitrary kernels. Thus, while the DGPS algorithm can use any differentiable kernel and does not encounter the complexities of variational methods, it loses the ability to integrate out the pseudo outputs.

With the addition of the pseudo data, we are required to learn the following set of parameters:

$$\boldsymbol{\Theta} = \left\{ \left\{ \bar{\boldsymbol{X}}_d^{(l)}, \bar{\boldsymbol{y}}_d^{(l)}, \boldsymbol{\theta}_d^{(l)} \right\}_{d=1}^{D^{(l)}} \right\}_{l=1}^{L}.$$

Now, by learning all the pseudo inputs and pseudo outputs, we lose some of the regularization advantages associated with GPs. With a larger set of parameters, overfitting poses more of a threat. However, by incorporating the prior into our model, the pseudo data is required to have a similar structure to the input data. In Chapter 5, we explore this advantage more in depth through an example.

4.3.5 General Algorithm

We now formally present the general Deep Gaussian Process Sampling algorithm. Consider an L-layer deep GP, consisting of one input layer $\boldsymbol{X} \in \mathbb{R}^{N \times D^{(0)}}$, one single-dimensional output layer $\boldsymbol{y} \in \mathbb{R}^N$, and $L-1$ hidden layers. Training involves learning two different types of parameters: the pseudo inputs and outputs, along with the hyper-parameters for every GP. Consider a GP connecting layer $l-1$ to the d'th hidden unit in layer l. We use $\boldsymbol{\theta}_d^{(l)}$ to denote the kernel parameters of this particular GP, $\bar{\boldsymbol{X}}_d^{(l)}$ to denote the pseudo inputs, and $\bar{\boldsymbol{y}}_d^{(l)}$ to denote the pseudo outputs. To be precise, $\bar{\boldsymbol{X}}_d^{(l)} \in \mathbb{R}^{M \times D^{(l-1)}}$, where M is the number of pseudo inputs (assumed to be the same for all GPs) and $D^{(l-1)}$ is the dimension of the previous layer (recall, this GP takes as input the previous layer values as a vector). Similarly, $\bar{\boldsymbol{y}}_d^{(l)} \in \mathbb{R}^M$.

Here, we provide pseudocode for sampling the hidden values at layer l, using the samples from the previous layer $\tilde{\boldsymbol{H}}^{(l-1)} \in \mathbb{R}^{N \times D^{(l-1)}}$. To sample values at layer $l = 1$, we define $\tilde{\boldsymbol{H}}^{(0)} = \boldsymbol{X}_*$, the input data for which we would like to generate samples. For hidden unit d in layer l, we are assuming

$$\tilde{\boldsymbol{H}}_{:,d}^{(l)} \sim \mathcal{N}(\tilde{\boldsymbol{\mu}}_d^{(l)}, \tilde{\boldsymbol{\Sigma}}_d^{(l)}) \tag{4.14}$$

where

$$\tilde{\boldsymbol{\mu}}_d^{(l)} = \boldsymbol{K}_{\tilde{\boldsymbol{H}}^{(l-1)}\bar{\boldsymbol{X}}_d^{(l)}} \boldsymbol{K}^{-1}_{\bar{\boldsymbol{X}}_d^{(l)}\bar{\boldsymbol{X}}_d^{(l)}} \bar{\boldsymbol{y}}_d^{(l)} \tag{4.15}$$

$$\tilde{\boldsymbol{\Sigma}}_d^{(l)} = \operatorname{diag}\left(\boldsymbol{K}_{\tilde{\boldsymbol{H}}^{(l-1)}\tilde{\boldsymbol{H}}^{(l-1)}} - \boldsymbol{K}_{\tilde{\boldsymbol{H}}^{(l-1)}\bar{\boldsymbol{X}}_d^{(l)}} \boldsymbol{K}^{-1}_{\bar{\boldsymbol{X}}_d^{(l)}\bar{\boldsymbol{X}}_d^{(l)}} \boldsymbol{K}_{\bar{\boldsymbol{X}}_d^{(l)}\tilde{\boldsymbol{H}}^{(l-1)}} \right). \tag{4.16}$$

Procedure 1 Sample $\tilde{\boldsymbol{H}}^{(l)}$ given $\tilde{\boldsymbol{H}}^{(l-1)}$

Require: $\tilde{\boldsymbol{H}}^{(l-1)}, \{\bar{\boldsymbol{X}}_d^{(l)}, \bar{\boldsymbol{y}}_d^{(l)}, \boldsymbol{\theta}_d^{(l)}\}_{d=1}^{D^{(l)}}$

1: $\tilde{\boldsymbol{H}}^{(l)}$:= Empty $N \times D^{(l)}$ matrix
2: **for** hidden unit d in $1, \ldots, D^{(l)}$ **do**
3: Calculate $\tilde{\boldsymbol{\mu}}_d^{(l)}$ using Equation 4.15
4: Calculate $\tilde{\boldsymbol{\Sigma}}_d^{(l)}$ using Equation 4.16
5: Sample $\tilde{\boldsymbol{h}}_d^{(l)} \sim \mathcal{N}(\tilde{\boldsymbol{\mu}}_d^{(l)}, \tilde{\boldsymbol{\Sigma}}_d^{(l)})$
6: $\tilde{\boldsymbol{H}}_{:,d}^{(l)} = \tilde{\boldsymbol{h}}_d^{(l)}$
7: **end for**
8: **return** $\tilde{\boldsymbol{H}}^{(l)}$

If we repeatedly sample values for every hidden layer, we can continue all the way to the final layer. To evaluate the model log-likelihood, rather than sampling from the final layer, we will use the final mean and covariance to compute the multivariate normal density. Thus, we introduce a function to sample the predictive mean and predictive variances from the final layer for an arbitrary input $\boldsymbol{X}_*$.

Procedure 2 Sample mean and covariance at final layer for input $\boldsymbol{X}_*$

Require: All parameters $\boldsymbol{\Theta}$

1: $\tilde{\boldsymbol{H}}^{(0)} := \boldsymbol{X}_*$
2: **for** layer l in $1, \ldots, L-1$ **do**
3: $\tilde{\boldsymbol{H}}^{(l)}$:= Sample $\tilde{\boldsymbol{H}}^{(l)}$ given $\tilde{\boldsymbol{H}}^{(l-1)}$
4: **end for**
5: Calculate $\tilde{\boldsymbol{\mu}}_1^{(L)}$ using Equation 4.15
6: Calculate $\tilde{\boldsymbol{\Sigma}}_1^{(L)}$ using Equation 4.16
7: **return** $\tilde{\boldsymbol{\mu}}_1^{(L)}, \tilde{\boldsymbol{\Sigma}}_1^{(L)}$

Finally, to compute our objective, we approximate the log-likelihood using Monte Carlo sampling. Additionally, we incorporate our prior over the pseudo parameters, returning the sum of the two terms.

Procedure 3 Evaluate model log-likelihood

Require: Training data $\boldsymbol{X}$, all parameters $\boldsymbol{\Theta}$

1: **for** k in $1, \ldots, K$ **do**
2: $\quad \tilde{\boldsymbol{\mu}}_k, \tilde{\boldsymbol{\Sigma}}_k :=$ Sample mean and covariance at final layer for input $\boldsymbol{X}$
3: **end for**
4: loglikelihood := $\text{logsumexp}\Big(\Big[\log P(\boldsymbol{y}|\tilde{\boldsymbol{\mu}}_1, \tilde{\boldsymbol{\Sigma}}_1), \ldots, \log P(\boldsymbol{y}|\tilde{\boldsymbol{\mu}}_K, \tilde{\boldsymbol{\Sigma}}_K)\Big]\Big) - \log N$
5: logprior := $\sum_{l=1}^{L} \sum_{d=1}^{D^{(l)}} \log P\left(\bar{\boldsymbol{y}}_d^{(l)} \big| \bar{\boldsymbol{X}}_d^{(l)}\right)$
6: **return** loglikelihood + logprior

For the above procedure, we note, of course, that

$$P(\boldsymbol{y}|\tilde{\boldsymbol{\mu}}_k, \tilde{\boldsymbol{\Sigma}}_k) = \mathcal{N}(\tilde{\boldsymbol{\mu}}_k, \tilde{\boldsymbol{\Sigma}}_k)$$

and

$$P(\bar{\boldsymbol{y}}_d^{(l)} | \bar{\boldsymbol{X}}_d^{(l)}) = \mathcal{N}\left(\mathbf{0}, \boldsymbol{K}_{\bar{\boldsymbol{X}}\bar{\boldsymbol{X}}}\right).$$

After evaluating our objective, we take the gradient with respect to $\boldsymbol{\Theta}$ to optimize our objective. To do this, we use automatic differentiation to ease computation on behalf of the user. The DGPS algorithm relies on *Autograd*, an automatic differentiation library in Python (Maclaurin et al., 2015). All that is required of us is to define our parameters $\boldsymbol{\Theta}$ and to provide a loss function. To optimize, we use the BFGS method with a fixed random seed. We can use BFGS because our objective is deterministic and, due to our random seed, we can evaluate the objective without stochasticity. There are many nice properties of using the BFGS algorithm for optimization, including approximate curvature and the fact that the user is not required to specify a step-size (Gill and Leonard, 2001).

4.3.6 2-Layer Example

As a concrete example, we now walk through the DGPS algorithm applied to a single-dimensional 2-layer deep GP.

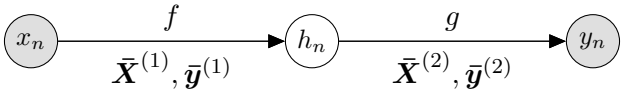

Figure 4.4: A two-layer deep GP with a single hidden dimension.

Consider again our 1-dimensional example, with a single hidden layer, as in Figure 4.4. We denote the GP connecting the first layer and hidden layer as f, and the GP connecting the hidden

layer and output layer as g. We would like to learn $\{(\bar{\boldsymbol{X}}^{(l)}, \bar{\boldsymbol{y}}^{(l)})\}_{l=1}^{2}$, the pseudo data for each layer, along with the kernel parameters for f and g, $\boldsymbol{\theta}^{(1)}$ and $\boldsymbol{\theta}^{(2)}$, respectively. (We drop the subscript denoting the hidden unit because each layer has dimension one.)

To start, we initialize the model parameters to random values. We would like to compute our objective, which consists of the model log-likelihood and the prior over the pseudo outputs. To evaluate the log-likelihood at our current parameter estimates, we first sample values from the hidden layer, $\boldsymbol{H}^{(1)}$, given our inputs, $\boldsymbol{X}$. Using the FITC approximation, we assume

$$P\left(\boldsymbol{H}^{(1)}|\boldsymbol{X}, \bar{\boldsymbol{X}}^{(1)}, \bar{\boldsymbol{y}}^{(1)}\right) = \mathcal{N}\left(\boldsymbol{\mu}^{(1)}, \boldsymbol{\Sigma}^{(1)}\right) \tag{4.17}$$

where

$$\boldsymbol{\mu}^{(1)} = \boldsymbol{K}_{\boldsymbol{X}\bar{\boldsymbol{X}}^{(1)}} \boldsymbol{K}^{-1}_{\bar{\boldsymbol{X}}^{(1)}\bar{\boldsymbol{X}}^{(1)}} \bar{\boldsymbol{y}}^{(1)} \tag{4.18}$$

$$\boldsymbol{\Sigma}^{(1)} = \operatorname{diag}\left(\boldsymbol{K}_{\boldsymbol{X}\boldsymbol{X}} - \boldsymbol{K}_{\boldsymbol{X}\bar{\boldsymbol{X}}^{(1)}} \boldsymbol{K}^{-1}_{\bar{\boldsymbol{X}}^{(1)}\bar{\boldsymbol{X}}^{(1)}} \boldsymbol{K}_{\bar{\boldsymbol{X}}^{(1)}\boldsymbol{X}}\right). \tag{4.19}$$

This is our standard conditional distribution using the FITC approximation, given by Equation 4.6. Note that although we are conditioning on the first-layer kernel parameters, $\boldsymbol{\theta}^{(1)}$, we drop these from the equation above for the sake of space. We now obtain K samples of the hidden layer values, $\{\tilde{\boldsymbol{H}}_k\}_{k=1}^{K}$, evaluated at our training data, by sampling from Equation 4.15. For each sample $\tilde{\boldsymbol{H}}_k$, we can approximate

$$P\left(\boldsymbol{y}|\tilde{\boldsymbol{H}}_k, \bar{\boldsymbol{X}}^{(2)}, \bar{\boldsymbol{y}}^{(2)}\right) \approx \mathcal{N}\left(\tilde{\boldsymbol{\mu}}^{(2)}, \tilde{\boldsymbol{\Sigma}}^{(2)}\right) \tag{4.20}$$

where

$$\tilde{\boldsymbol{\mu}}^{(2)} = \boldsymbol{K}_{\tilde{\boldsymbol{H}}_k\bar{\boldsymbol{X}}^{(2)}} \boldsymbol{K}^{-1}_{\bar{\boldsymbol{X}}^{(2)}\bar{\boldsymbol{X}}^{(2)}} \bar{\boldsymbol{y}}^{(2)} \tag{4.21}$$

$$\tilde{\boldsymbol{\Sigma}}^{(2)} = \operatorname{diag}\left(\boldsymbol{K}_{\tilde{\boldsymbol{H}}_k\tilde{\boldsymbol{H}}_k} - \boldsymbol{K}_{\tilde{\boldsymbol{H}}_k\bar{\boldsymbol{X}}^{(2)}} \boldsymbol{K}^{-1}_{\bar{\boldsymbol{X}}^{(2)}\bar{\boldsymbol{X}}^{(2)}} \boldsymbol{K}_{\bar{\boldsymbol{X}}^{(2)}\tilde{\boldsymbol{H}}_k}\right). \tag{4.22}$$

Thus, we can approximate the marginal likelihood with our samples:

$$P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\Theta}) \approx \frac{1}{K}\sum_{k=1}^{K} P\left(\boldsymbol{y}|\tilde{\boldsymbol{H}}_k, \bar{\boldsymbol{X}}^{(2)}, \bar{\boldsymbol{y}}^{(2)}\right). \tag{4.23}$$

As the number of samples K grows large, our approximation approaches the true value. Additionally, we would like to evaluate the log-likelihood as opposed to the likelihood to avoid numerical instability issues. Thus, we approximate the log-likelihood as

$$\log P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\Theta}) \approx \text{LSE}\left(\log P(\boldsymbol{y}|\tilde{\boldsymbol{H}}_1, \bar{\boldsymbol{X}}^{(2)}, \bar{\boldsymbol{y}}^{(2)}), \ldots, \log P(\boldsymbol{y}|\tilde{\boldsymbol{H}}_K, \bar{\boldsymbol{X}}^{(2)}, \bar{\boldsymbol{y}}^{(2)})\right) - \log N \tag{4.24}$$

where LSE is the LogSumExp function.

Finally, we incorporate the prior over the pseudo outputs into our objective. Thus, we define the following objective:

$$L(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\Theta}) = \log P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\Theta}) + \sum_{l=1}^{L} \sum_{d=1}^{D^{(l)}} \log P\left(\bar{\boldsymbol{y}}_d^{(l)} \middle| \bar{\boldsymbol{X}}_d^{(l)}\right). \tag{4.25}$$

By maximizing the above function with respect to $\boldsymbol{\Theta}$ using automatic differentiation to evaluate gradients, we can learn the parameters $\boldsymbol{\Theta}$ of our model.

4.3.7 Testing

Once we have learned our parameters $\boldsymbol{\Theta}$, we would like to evaluate our model on a set of new inputs $\boldsymbol{X}_*$. If we have outputs $\boldsymbol{y}_*$ that correspond to these inputs, we can evaluate the test log-likelihood in the same way as the train log-likelihood. That is, we sample hidden values at every layer, until we have a single predictive mean and predictive covariance for the output layer. We can then average the log-likelihood at every sample to approximate the test log-likelihood of our model.

To make predictions, we note that if we sample values all the way to the final layer, our samples are not normally distributed. Rather, our samples represent a mixture of Gaussians. Therefore, there are a few ways to make predictions. We can take any one sample and use it to make predictions at every new input, since we are sampling from a mixture of Gaussians. Additionally, we can find the Gaussian that best approximates this mixture by matching moments. Finally, a third way to make straightforward predictions is to predict by averaging all of our samples.

We note that the FITC approximation assumes that values from the training data are independent given the pseudo data. The original paper by Snelson and Ghahramani (2005) does not provide insight into whether this assumption goes away for the test data. Therefore, we can either

carry the assumption or drop it; the only difference when we drop is that whenever we would like to compute the predictive covariance, we use the entire matrix instead of the diagonal. Throughout this thesis, we drop the approximation for test data, largely to produce smoother and more interpretable predictions.

4.3.8 Implementation Notes

Below, we note a few details that are required to successfully implement the DGPS algorithm.

In Section 4.3.3, we described how to take derivatives of samples from a normal distribution. Extending to the multivariate case, recall from Chapter 2 that we can represent $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ as $\boldsymbol{x} = \boldsymbol{V}\boldsymbol{z} + \boldsymbol{\mu}$ where $\boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{\Sigma}$ and $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. Thus, whenever we would like to evaluate a gradient that relies on samples, we sample a multivariate standard normal and apply the above transformation to obtain a sample from our desired distribution.

This parameterization relies on finding a decomposition of $\boldsymbol{\Sigma}$ into the matrix $\boldsymbol{V}$. During training, $\boldsymbol{\Sigma}$ is assumed to be diagonal because of the FITC approximation. Therefore, we take the square root of every entry on the diagonal. If, during testing, we no longer assume the predictive distribution is conditionally independent given our training data, we can use the Cholesky decomposition, which finds a unique solution because $\boldsymbol{\Sigma}$ is always assumed to be positive definite.

Occasionally, we run into numerical stability issues, where, due to poor float precision, a covariance matrix is not recognized as positive definite. We fix this by adding the constant 10^{-6} to the diagonal. This clears up the float precision issues and keeps the matrix positive definite, since the eigenvalues uniformly increase by 10^{-6}. Moreover, this does not change the structure of the matrix drastically because the constant is small in magnitude.

Chapter 5

Experiments and Analysis

In this section, we explore how the Deep Gaussian Process Sampling algorithm fits data sets – both toy and real – and discuss the properties of different aspects of our algorithm. How does the number of samples used to approximate the marginal likelihood affect the computational complexity? How does the predictive accuracy change when we increase the number of layers of the deep Gaussian process? We first explore these questions in the context of fitting a step function, as it illuminates many interesting qualities and challenges of the DGPS algorithm. Next, we explore another toy data set with non-stationary data, where the shape of our function depends on the region of the input space. We finish by fitting a real-world data set with motorcycle accident data. These examples affirm that while deeper architectures are better suited models to learn non-stationary data, the DGPS algorithm objective becomes more prone to local optima as the number of layers grows. This is a fundamental weakness of the DGPS algorithm, and it can be softened by using random restarts at different parameter initializations.

5.1 Step Function

The first data set we test is a step-function with noise. That is, $\boldsymbol{X} \in \mathbb{R}^{N \times 1}$, and $y_i = \text{sign}(x_i) + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. Specifically, we use inputs x_i uniformly distributed between -2 and 2, and $\epsilon = .01$. We choose the step-function because its non-stationarity is appealing from a deep GP perspective (Damianou, 2015). While a single-layer GP would require a non-stationary kernel to fully capture the fluctuations of the function from $y = -1$ to $y = +1$, ideally a deep GP would

capture the flatness at each step, along with the steep incline between them. Additionally, we prefer models that offer reasonable predictive hypotheses beyond the tails of our input data. That is, we do not know how the function should behave at points outside the range of our training values, but it is reasonable to assume it will fluctuate between the two steps. As a result, we are interested in observing the variance of our predictive draws for a wide range of inputs.

5.1.1 Qualitative Comparisons

Consider a standard single-layer GP with the squared exponential kernel fit on this step function by optimizing the marginal likelihood with respect to l^2, σ_f^2, and σ_n^2. As a reminder, this kernel corresponds to

$$k(x, x') = \sigma_f^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right) + \sigma_n^2 I(x = x') \tag{5.1}$$

where $I(\cdot)$ is the indicator function. In Figure 5.1, we plot samples from the predictive distribution, along with the predictive mean and predictive covariance.

Evidently, the predictive draws do not fully capture the shape of the step function. We do not see flat line segments at $y = -1$ or $y = +1$; rather, the functions curve as they fit the data closely. The learned length-scale is quite small, at $l^2 = .329$, so the predictive draws become uncorrelated quickly in the input domain. Outside the range of inputs, the functions fluctuate rapidly, and do not return to the steps. Rather, they extend significantly beyond the steps, both above and below – this poorness is due to the squared exponential covariance function being stationary. Because the step function requires learning a steep rise at a certain range of inputs along with flatness at other portions of the input space, the squared exponential kernel is not ideally suited to capture these fluctuations in the single-layer case.

How do deeper models affect predictive performance? In Figure 5.2, we plot draws from the predictive distribution from the 2- and 3-layer deep GP architectures for the noisy step function, comparing them to the single-layer GP. It is important to note that these predictive draws are dependent on the parameter initializations and locations of the original data. In the Experiments section, we explore instances where, due to the difficulties posed by optimizing with respect to a large parameter space, deeper models get stuck during optimization.

The draws from the predictive distribution for the single-layer GP reflect Figure 5.1, indicating

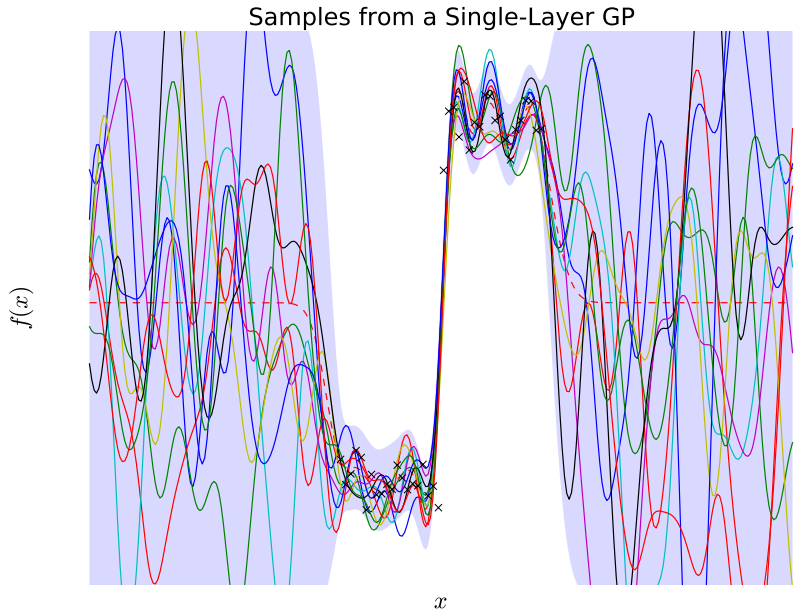

Figure 5.1: Functions sampled from a single-layer GP with the squared exponential kernel trained on a step-function with noise. We use $N = 40$ training points, which are marked by 'x', while the predictive mean is depicted by the dashed red line, and the 95% predictive confidence interval is shaded in purple. After optimizing the log marginal likelihood, we learn $\sigma_n^2 = .041, \sigma_f^2 = .666$, and $l^2 = .329$. To plot the functions, we create a range of values $\boldsymbol{X}_*$ that spread beyond the input space, calculate the predictive mean $\boldsymbol{\mu}_*$ and predictive covariance $\boldsymbol{\Sigma}_*$ at these inputs, and draw corresponding $\boldsymbol{y}_* \sim \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$. Note that we do not use pseudo data or the FITC approximation. Best viewed in color.

that a single-layer model is unable to capture the smoothness at both steps of the function. The 2-layer deep GP, by comparison, does a better job. For one, the incline between the two steps is steeper, and it is also somewhat flatter at the steps. Additionally, it puts more weight at the tails on the step values, $y = -1$ and $y = +1$. We note that some draws for the 2-layer case fluctuate from these values and vary wildly in the output domain, but they are more solidly around -1 and $+1$ than in the 1-layer case. Finally, the 3-layer deep GP features the most significant incline, and the tail behavior captures the steps more firmly than the two shallower architectures. This is our desired behavior – intuitively, we are uncertain about function values at points outside the input domain, but reasonably, we believe they should hover around the two step values, $y = -1$ and $y = +1$.

To understand the behavior of the 2-layer deep GP, we turn to the right-hand column of Figure 5.5, which displays predictive draws from each layer. In the first layer, which maps x to the hidden values, the pseudo inputs appear to divide into three clusters of equal size. It appears that two

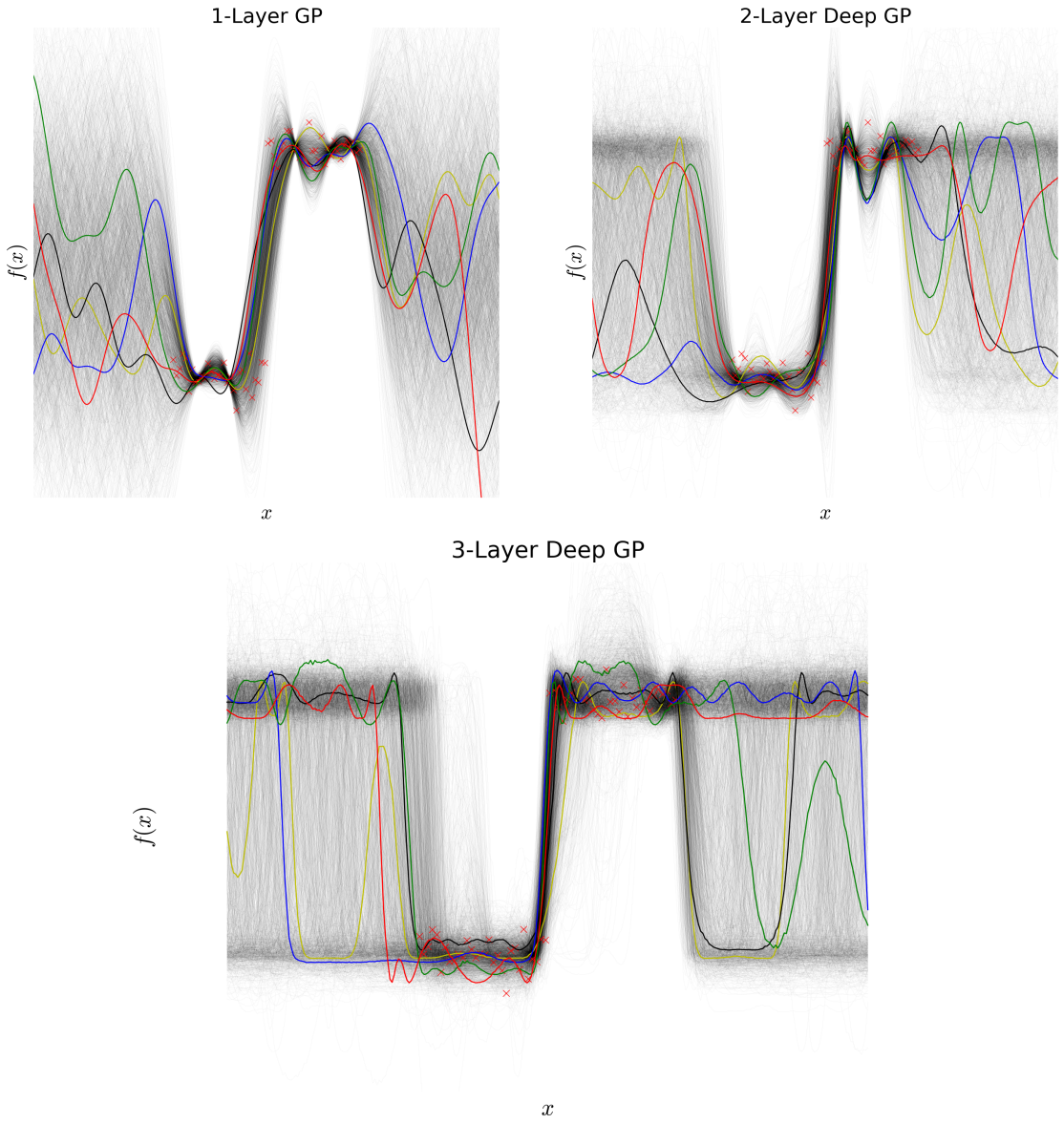

Figure 5.2: Predictive draws from a single-layer GP, a two-layer deep GP, and a three-layer deep GP trained on the noisy step function. We use $N = 60$ data points, each point depicted as a red 'x', along with 10 pseudo data points for each layer in each process. The plots represent 2000 draws from each predictive distribution, with 5 displayed in color. As the number of layers grows, the architecture is better suited to fit the non-stationary step function, as the incline between steps becomes steeper, and the tail behavior puts more weight toward $y = -1$ and $y = +1$.

clusters correspond to the two steps in the original data, and the intermediate cluster corresponds to the incline between the steps. Although the curves do not fit the pseudo data well, they appear to correspond to a gradual incline between steps.

Meanwhile, the second layer produces much more varied functions, corresponding to a smaller length-scale. Here, the pseudo data are in two clusters, each of which corresponds to a step. Additionally, due to the smaller length-scale, the incline between steps is much steeper. Inputs that are close to 0 are mapped to the incline in each layer and thus to the incline in the composition of the two functions as well. The other training inputs get mapped to the incline in the first layer but not in the second layer, causing them to belong to one of the two steps. Finally, the inputs outside the range of the training input space get mapped to the constant areas of both GPs, so their mean approaches $y = 0$ in the full composition.

Judging by these qualitative results, deeper architectures appear better-suited to learn the step function. We compare these models in more depth empirically in the Experiments section. Moving forward, we focus on the necessity of the pseudo output prior in conjunction with the role of parameter initializations when using the DGPS algorithm.

5.1.2 Prior Analysis

In Chapter 4, we discussed how the prior over pseudo outputs was necessary to prevent overfitting. The prior encourages the pseudo data to resemble the training data because they share the same kernel, so the risk of pseudo outputs taking on extreme values decreases.

Training a single-layer GP with pseudo data on the noisy step function illustrates the necessity of this prior. Again, consider $N = 40$ training data points, and a single-layer GP with the squared exponential covariance function. We also introduce 10 pseudo inputs and their corresponding outputs. Using the log marginal likelihood of the data as our objective function, without any prior on the pseudo data, we obtain predictive draws as shown in Figure 5.3.

Here, the pseudo inputs appear to approximate the training data, as they spread out evenly across the input domain. However, the pseudo outputs stray quite far from the outputs in the training data. Because there is no prior used for the location of the pseudo outputs, the shape of the pseudo data does not match that of the training data. Rather, the pseudo outputs take on the values that will increase the marginal likelihood, which can result in overfitting. Indeed, the mean

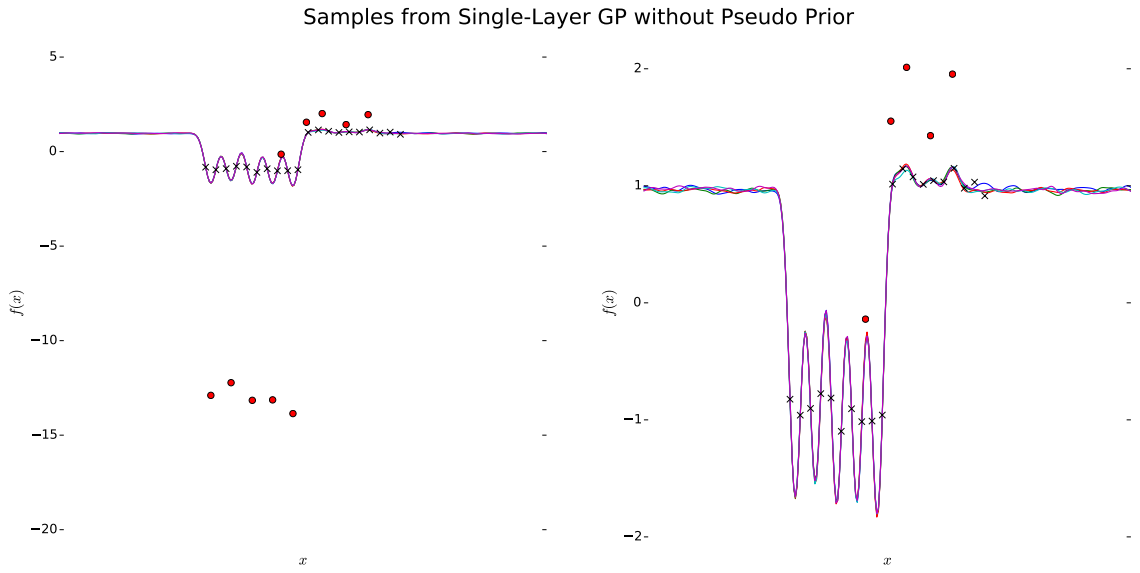

Figure 5.3: Draws from the predictive distribution of a single-layer GP trained on the noisy step function with 10 pseudo data points, without putting a prior on the pseudo data. The pseudo data points are illustrated by red circles and the training data is denoted by the letter 'x'. The two graphs display the same results; the graph on the right is zoomed in further to reveal the degree to which we overfit. The learned parameters are $\sigma_n^2 = .001, \sigma_f^2 = .0002$, *and* $l^2 = .121$. *The mean squared error of the training data is* 0.0003.

squared error of the training data is a mere .0003, so it learns the training data almost exactly.

By placing a prior on the pseudo outputs, we encourage the pseudo data to resemble the training data, as they share the same kernel. As a reminder, if the pseudo data consists of $\bar{\boldsymbol{X}}$ as the pseudo inputs and $\bar{\boldsymbol{y}}$ as the pseudo outputs, we place the following prior on the outputs:

$$P(\bar{\boldsymbol{y}}|\bar{\boldsymbol{X}}) = \mathcal{N}\left(\mathbf{0}, \boldsymbol{K}_{\bar{\boldsymbol{X}}\bar{\boldsymbol{X}}}\right).$$

Figure 5.4 depicts the predictive draws for the single-layer model, using the prior described above. As we can see, the pseudo outputs no longer take on extreme values outside the range of our output space. Instead, they are concentrated around $y = -1$ and $y = +1$. Additionally, the predictive draws are more varied, which better resemble our uncertainty at the tails of the step function, outside the range of the input space. Clearly, overfitting is no longer as much of an issue, as the predictive draws yield sensible fits that approximate a step function, albeit imperfectly.

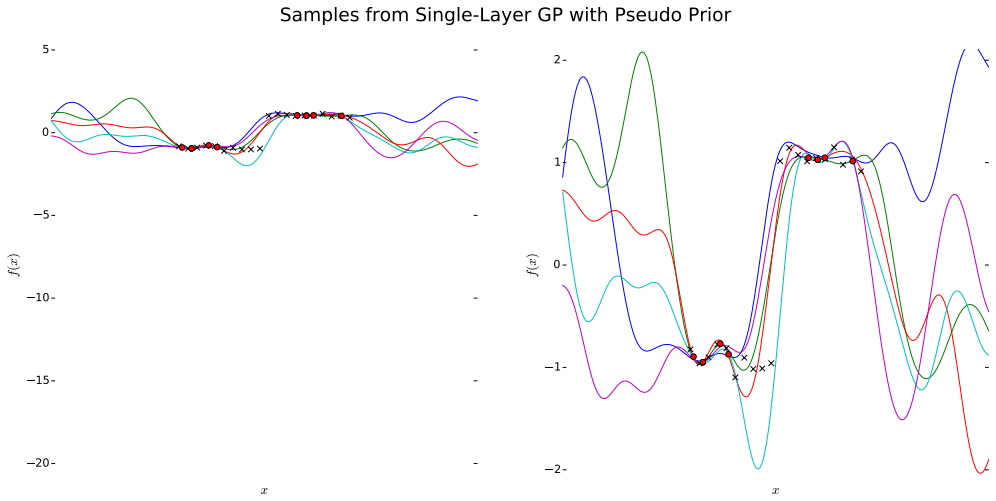

Figure 5.4: Again, draws from the predictive distribution of a single-layer GP trained on the noisy step function with 10 pseudo data points, this time with a prior on the pseudo outputs. The graphs have the same dimensionality as Figure 5.3; now, the predictive functions are more varied, and they do not overfit as the pseudo data spreads out less. The learned parameters are $\sigma_n^2 = .001, \sigma_f^2 = .763$, *and* $l^2 = 1.69$. *The mean squared error of the training data is now* 0.072*, indicating a good fit without necessarily overfitting.*

5.1.3 Initialization

Additionally, parameter initializations play an important role in the DGPS algorithm. Consider a 2-layer model, where the hidden layer is single-dimensional, with the squared exponential kernel. If we randomly initialize all parameter values, we risk encountering local maxima in our objective function that do not fit the data well. For example, if the length-scale parameter for any layer is initialized to a value too small, this may lead to a local maximum where the full function composition can remain highly varying.

Instead of a random initialization, we would like to incorporate some knowledge we have about our data and the model to obtain a smart initialization. It is easy to be cautious about the length-scales; because we risk getting stuck in highly-varying functions when the length-scales are too low, for all layers besides the first, we initialize them to be large, at about 10 times the variance of our data. For the first layer, which is directly linked to the training data, we choose the length-scale to be the median distance between all pairs of input data points, which has been empirically shown to provide a reasonable fit (Houlsby et al., 2012).

Additionally, the pseudo data can be initialized in a smart fashion as well. For the first layer,

we would like the pseudo inputs to approximate the data. For K pseudo inputs, one option is to randomly sample (without replacement) K points from the training data to use as our first-layer pseudo inputs. However, an even smarter procedure is initializing with K-Means, choosing K clusters and using the cluster centers as our pseudo inputs. These K clusters should provide a reasonable way to distribute the first-layer pseudo inputs over the range of possible input values. For all layers besides the first, we have no initial knowledge about the pseudo data other than that the outputs should be reasonably similar to the inputs. Therefore, we draw the pseudo inputs from a normal distribution and set the pseudo outputs for each GP to be equal to the pseudo inputs.

In Figure 5.5, we plot draws from the predictive distribution of a trained 2-layer deep GP, one with the random initialization and one with the smart procedure. While the differences may appear small, they are indeed substantial – the trained 2-layer GP with the smart initialization produces a fit that is less jagged and fluctuates less between the steps. Additionally, the training log-likelihood is 38.3 for the random initialization, compared to 48.6 for the smart initialization. The predictive draws with the random initialization do not capture the bottom step well, and appear to be less smooth than the smart initialization. These disadvantages are likely due to the small length-scales and to the fact that the pseudo data does not spread out for each layer.

5.1.4 Experiments

To empirically test the success of our models for the noisy step function, we run experiments comparing prediction quality for varying aspects of the DGPS algorithm, including: the number of layers in the model, the size of the training data, the number of samples we use to approximate the model likelihood, and the size of our pseudo dataset.

We first vary the number of layers. We take 80% of the data as our training set, and use the remaining 20% to test. Then, for 1-, 2-, and 3-layer deep GPs (where each hidden layer has one unit), we train by running the DGPS algorithm on our training data and then evaluate on our test set. We compute the test log-likelihood by taking 100 samples of output means and covariances for a particular set of inputs, and average the log probability of the test outputs given these samples. We then divide this value by the number of points in our test set, to compute log-likelihood per data. To make predictions, we average the 100 sampled means and measure mean squared error with the true values. We do this for varying sizes of our data set: $N = 50, 100$, and 200 (note that

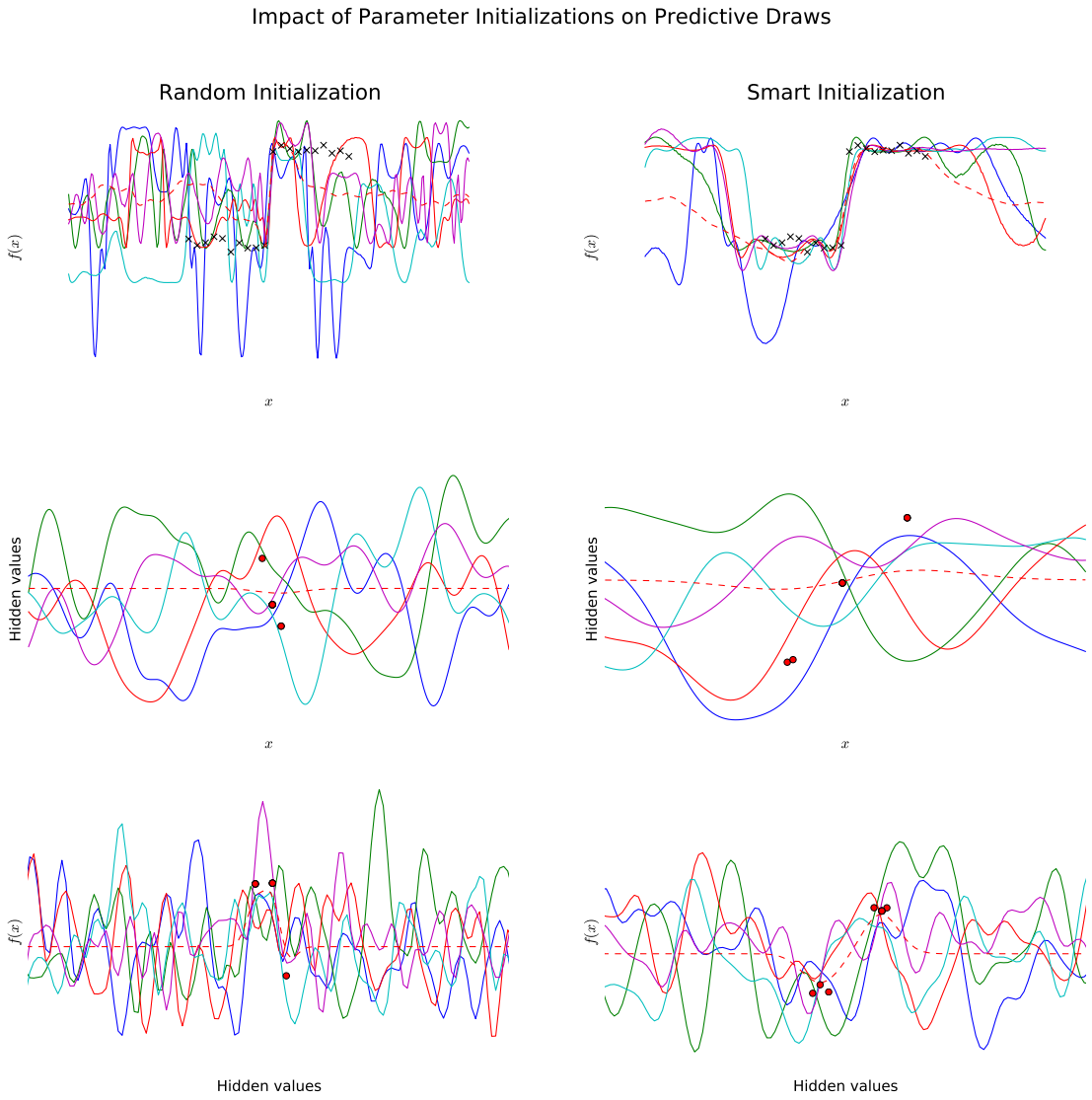

Figure 5.5: Two examples of a two-layer deep GP trained on the noisy step function; on the left, we initialize randomly, and on the right, we initialize with the smart initialization detailed in Section 5.1.3. The top row displays samples from the trained deep GP for both initialization methods. The middle row represents samples from the input space to the hidden space, while the bottom row represents samples from the hidden space to the output space. The red dashed line for each layer represents the predictive mean (for the full GP, it is the mean of the sampled means), and the location of the pseudo data for the individual layers is denoted by red circles. The training log-likelihood for the random initialization is 38.3*, compared to* 48.6 *for the smart initialization.*

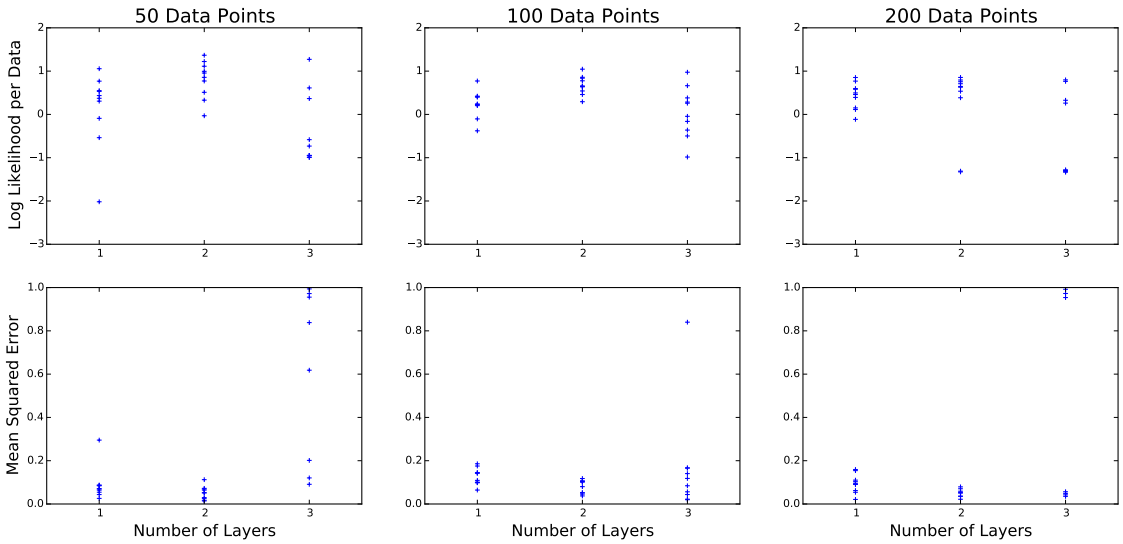

Figure 5.6: Experimental results measuring the test log-likelihood per data and test mean squared error of the noisy step function. We vary the number of layers used in the model, along with the number of data points used in the original step function (which is divided 80/20 into train/test). We run 10 trials at each combination, changing the points used to train and test.

from these values, 80% is used as our training set, and 20% for our test set). For all models, we use 10 pseudo data points and 10 samples to approximate the log-likelihood. Finally, we run 10 trials of each experiment, running our optimizer for a maximum of 1,000 iterations. The results are plotted in Figure 5.6.

It is not immediately clear from this diagram that increasing the number of layers improves our predictions, using either mean log-likelihood or mean squared error as a metric. As an example, consider our results using 100 data points. The distribution of test mean squared error for the 2-layer model appears to be shifted lower than the results for the 1-layer model, albeit not significantly. However, our results for three layers are more highly varying – at its best, the model has a smaller test mean squared error than 1- or 2-layer models, yet it also produces the experiment with by far the worst mean squared error, .84, compared to a second-worst of .186 from the 1-layer model. Additionally, .84 represents a somewhat abysmal mean squared error – recall, the outputs of the step function are centered at around $+1$ and -1, so the predictions for this model are quite uninformative. Even removing this particular outlier, the spread of errors reflects that of the 1-layer model, so it is not clear which model is superior when using mean squared error as a metric. The log-likelihood results reflect similar findings, with two layers providing a marked improvement over

one layer, although the positive trend does not extend to three layers.

These findings appear to reflect the idea that deeper architectures do not necessarily provide better predictions, regardless of the number of data points used in the experiments. The 3-layer model occasionally has trials with the highest test log-likelihood, yet it also has the widest distribution and trials with the worst results. Thus, if we were to plot log-likelihood averages, the 3-layer means would be lower than those of 2-layers, and occasionally lower than the single-layer results. Theoretically, deeper architectures should be able to fit anything a single-layer GP can, by learning the identity at all but one layer.

An initial hypothesis to explain these findings is that we are overfitting during training. Since the model is required to learn more parameters as the number of layers increases (both pseudo data and kernel parameters), we are making it easier to fit the training data exactly. By learning pseudo parameters for every layer, are we losing the regularization advantages associated with Bayesian models?

To answer this question, we plot the training set log-likelihood per data against the test set log-likelihood per data in Figure 5.7. We do the same for mean squared error. These plots depict a linear trend. Specifically, if a model does not make accurate predictions on the test set, it is not a result of overfitting – rather, it is a result of not fitting the training data well enough. If overfitting were a factor, the training log-likelihood would be much higher than that of the test set. While the predictions appear to systemically decline from training to testing, these do not explain the outliers in Figure 5.6; for example, all the points with test log-likelihoods lower than -1 also had train log-likelihoods lower than -1, most of them coming from the 3-layer architecture.

Thus, overfitting does not appear to pose a problem as the architecture of our model increases. Rather, our results indicate that if we can successfully optimize our objective, deeper architectures are better suited at learning the noisy step function than simpler ones. The caveat, then, is that it becomes more difficult to train and successfully optimize as the number of layers grows and the number of parameters increases.

As an example, we present a three layer architecture trained on the noisy step function with $N = 200$ data points. The two plots in Figure 5.8 represent the same model – the only difference is that they use different random seeds to initialize the parameters. These random seeds produce drastically different fits. With the random seed on the left, our training log-likelihood is 186.9 and

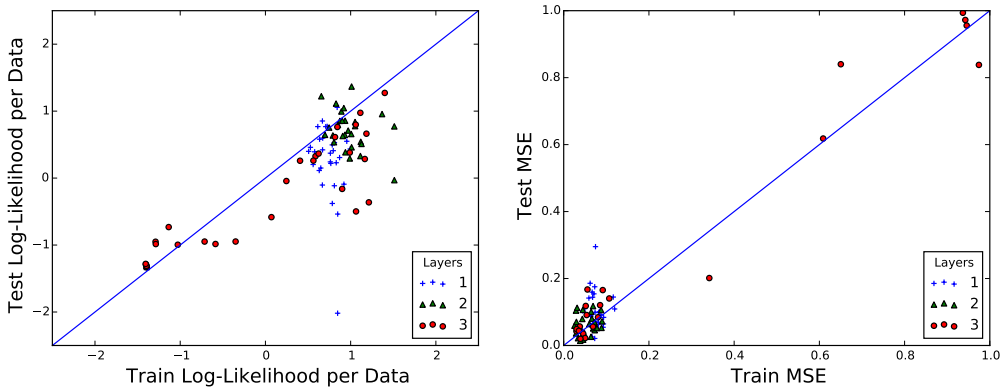

Figure 5.7: Test set log-likelihoods per data and mean squared errors plotted against their training set counterparts for the step function experiment. We plot the identity (in blue) for the sake of comparison.

our training MSE is 1.01. The optimization routine gets stuck in a local maximum with a poor fit. Because the first two layers are flat (corresponding to large length-scales), all the variance in the final set of predictions stems from the final layer, which is spiky. As a result, the composition of the functions matches the final layer. Meanwhile, the initialization on the right yields a training set log-likelihood of 174.1, and a corresponding MSE of 0.04. The GPs in each hidden layer also feature less extreme length-scales, which are better suited to fit the corresponding pseudo parameters.

These plots support the notion that deeper architectures can be more difficult to train. However, the experimental results from Figure 5.6, along with the plots of the predictive densities in Figure 5.2, reaffirm that, if properly trained, deeper architectures can be advantageous for fitting complex functions.

How then, can we make the optimization easier? The smart initialization described in Section 5.1.3 provides a step in the right direction. By initializing parameters close to reasonable values, the risk of getting stuck in an unsatisfactory local maximum decreases. Additionally, we have noted that our fit is a good indicator of predictive accuracy. Therefore, every time we train, we can optimize with respect to different random seeds. We can then evaluate which of these seeds produces the best fit, measured in training log-likelihood or training MSE, using that particular parameterization in our predictive model. This method, known as *random restarts*, provides one method to guard against problems posed by optimization difficulties.

In Figure 5.9, to demonstrate the random restarts technique, we record the test log-likelihood for the trial at each combination that had the highest training log-likelihood. That is, using the 10

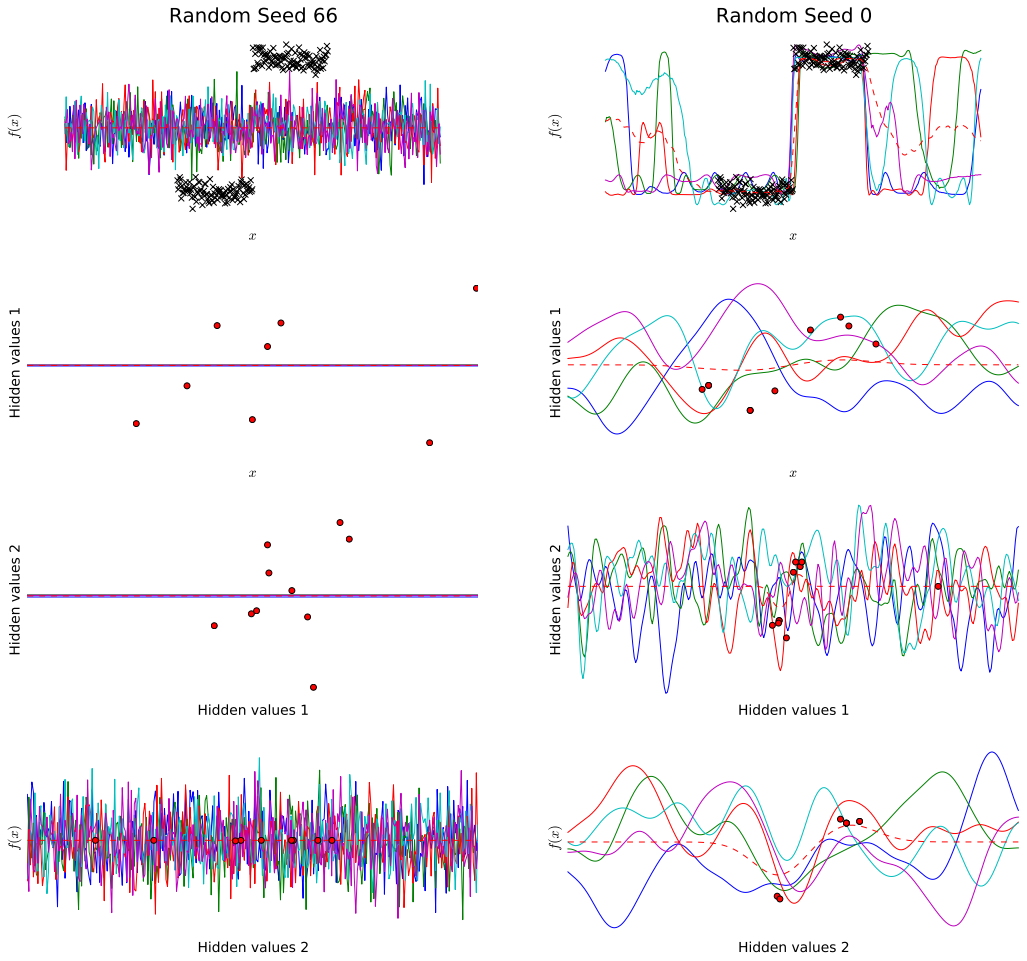

Figure 5.8: Predictive draws from two identical three-layer models, albeit with different random parameter initializations. The initialization on the left encounters a local optimum where two layers learn flat functions. Meanwhile, when we optimize using the parameter initialization on the right, we better capture the shape of the noisy step function data.

trials at each setting of the experiment, we take the trial with the highest training log-likelihood and display the corresponding test log-likelihood. As we can see, the test log-likelihood appears to increase as the number of layers increases, as the 3-layer model has the largest test log-likelihood using 50 and 200 data points, while the 2-layer model has the largest test log-likelihood with 100 data points. These results appear to coincide with the notion that, when successfully trained, deeper models are better suited to fit the step function.

How do the number of pseudo data points and the number of samples used to approximate the likelihood impact the quality of our predictions? To test these effects, we create a noisy step function data set with 125 points, again using an 80/20 train/test split. For all combinations, we

Number of Data Points	1 Layer	2 Layers	3 Layers
50	.44	−.03	**1.27**
100	.23	**.54**	−.36
200	−.11	.71	**.80**

Figure 5.9: The test log-likelihood per data corresponding to the highest training log-likelihood for each combination of layers and data points. The largest number in each row is bolded.

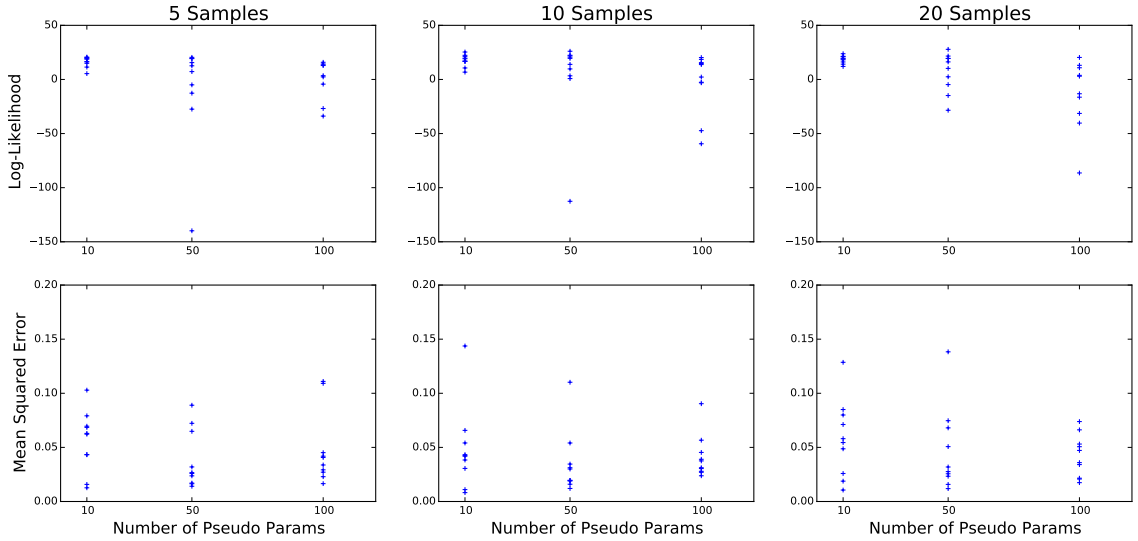

Figure 5.10: Experimental results measuring the test log-likelihood and test mean squared error of the noisy step function. We vary the number of samples used in the Monte Carlo steps of our model, along with the number of pseudo parameters. We use a 2-layer architecture, running 10 trials at each combination.

use a 2-layer model. We compare the results when using 5, 10, and 20 samples to approximate our objective, and for each sample size, we test the results using 10, 50, and 100 pseudo data points. We then record the test set log-likelihood, mean squared error, and training time, again running 10 iterations of each experiment. The results are displayed in Figure 5.10.

Theoretically, we would expect the model to generate better predictions when the number of points in our pseudo data set increases. However, there does not appear to be a clear positive trend in the results plotted in Figure 5.10. If we look at the plot for 5 samples, the range of log-likelihoods becomes larger as the number of pseudo parameters increases, although they all top out at about the same amount. This trend is even more marked for 20 samples. The range of log-likelihood values expands downwards as the number of pseudo parameters grows.

There are a couple of ways to explain these trends. One possibility is that the particular data set being trained does not require many pseudo parameters to learn, so the variance in our prediction

success is due to randomness. Another possibility, which is more plausible, is that these results coincide with our findings from before, namely, that more parameters make the model harder to optimize. Recall that the pseudo data is used at every layer, so the number of additional parameters to learn scales by a multiplicative factor as the number of pseudo parameters increases.

Thus, it appears that in practice, additional pseudo parameters do not improve the predictive performance of a deep GP. Rather, they make optimization more difficult. Of course, presumably there is a threshold before which additional pseudo parameters become more helpful; for example, it would be very hard to fit a data set with one pseudo input and output. However, after this threshold, the addition of pseudo data makes our model more complicated, forcing us to optimize with respect to a more complex parameter space.

We can also use the plots in Figure 5.10 to assess the impact of increasing the number of samples in our Monte Carlo step. Our predictions do not appear to improve as we use more samples to approximate the log marginal likelihood. There does not appear to be a negative effect either – rather, there is not a clear correlation between accuracy and number of samples, at least past 5 samples. Consider the mean duration of training with respect to a varying number of samples, depicted in Figure 5.11. Doubling the number of samples from 5 to 10, and then again from 10 to 20, doubles the average training duration for both instances. This should not come as a surprise, as each sample requires propagating extra values through the entire model architecture. Additionally, the presence of more pseudo parameters adds to this training time. Because the number of samples does not correlate directly to predictive performance, for computational purposes, it may not make sense to use more than 5 in practice.

Besides using random restarts and decreasing the number of model parameters, a possible way to combat the challenges posed by optimization is to use a different optimization procedure. In these experiments, we use the BFGS optimization routine. One possible avenue to explore would be using first-order gradient-based optimization methods. Although this would require the user to experiment with learning rates, a method such as Adam, which tunes parameters at different rates, may help avoid local optima (Kingma and Ba, 2014). Additionally, our models in these experiments use one unit in each hidden layer. It would be beneficial to experiment with more diverse architectures, in case increasing the dimensions of the hidden layers makes optimization easier.

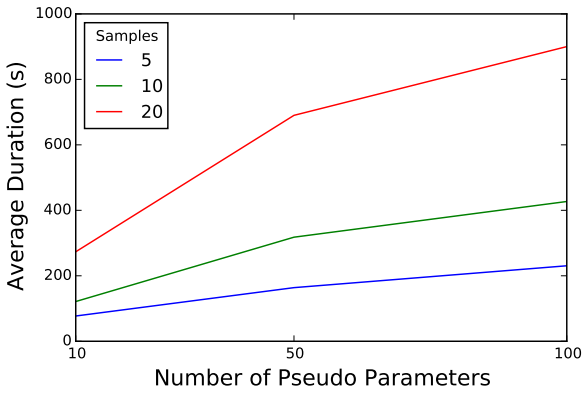

Figure 5.11: The average duration of training, plotted against the number of samples used to approximate the likelihood and the number of pseudo parameters present in the model. The results are from a 2-layer deep GP trained on a noisy step function, using a maximum of 1000 iterations in the BFGS optimization.

5.2 Toy Non-Stationary Data

To evaluate a deep GP's ability to learn non-stationary functions, we create toy non-stationary data and fit it with various architectures. Specifically, we create one-dimensional input data $\boldsymbol{X} \in \mathbb{R}^{120\times 1}$, where each input x_i is between -4 and 4. We divide the input space into three regions, $\boldsymbol{X}_1 \in [-4, -3]$, $\boldsymbol{X}_2 \in [-1, 1]$ and $\boldsymbol{X}_3 \in [2, 4]$, each of which consists of 40 data points. For each region, we sample corresponding outputs from a GP using the squared exponential kernel with signal variance 1. However, for region $\boldsymbol{X}_1$ and $\boldsymbol{X}_3$, we use length-scale $l = .25$, while for region $\boldsymbol{X}_2$ we use length-scale $l = 2$. This results in a data set where the tails are spiky and highly-varying, while the center is more smooth.

In Figure 5.12, we use a single-layer GP and a 2-layer deep GP to fit the data. The 2-layer model has a 1-dimensional hidden layer, and both models use 50 pseudo data points. Additionally, we standardize the data to improve numerical precision.

Similar to the step function results from Section 5.1, the predictive draws from the 2-layer model are better suited to capture the non-stationarity of the data. The one-layer model recovers only one length-scale, which, at $l = .461$, is close to the value used to sample function draws for $\boldsymbol{X}_1$ and $\boldsymbol{X}_3$, .25. Thus, while it captures the curvature of the data in these regions, the predictive draws do not flatten out for the intermediate region. Meanwhile, the 2-layer model benefits from its ability to compose two different length-scales with one another. It is noteworthy that the recovered length-

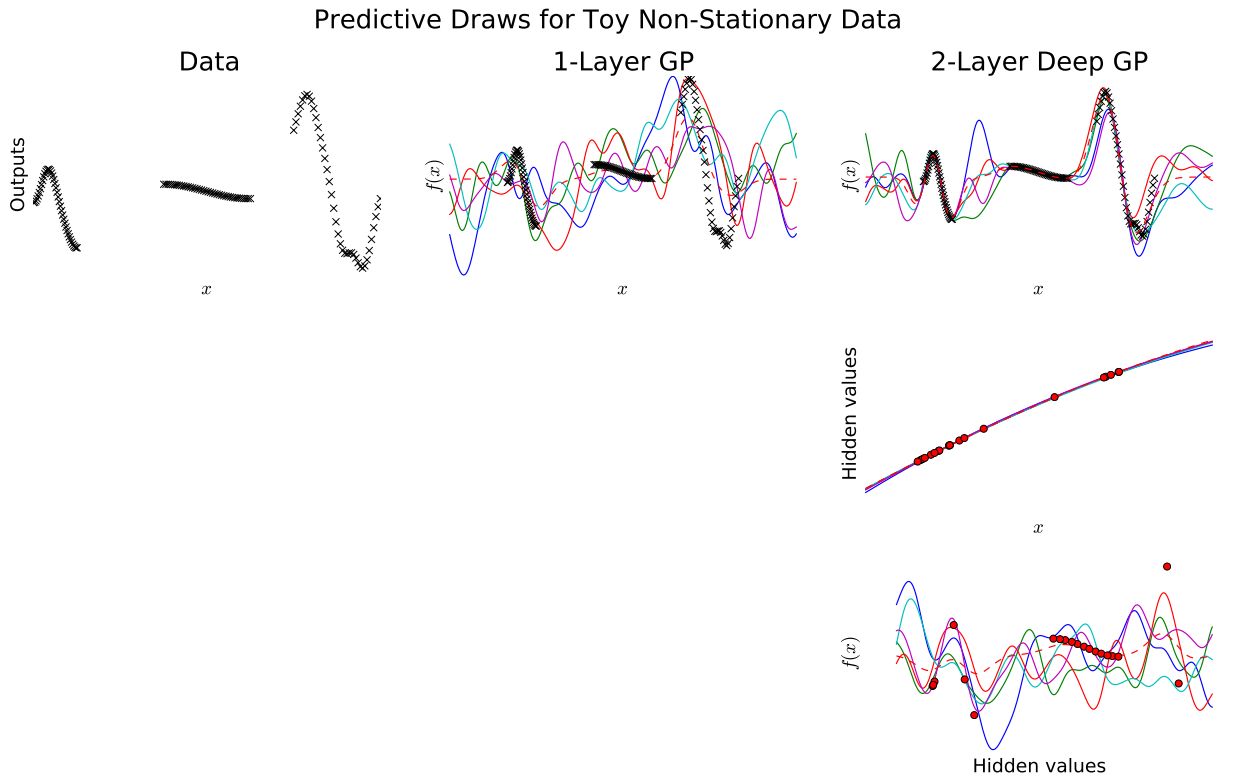

Figure 5.12: Predictive draws from the single-layer and 2-layer models for toy non-stationary data with squared exponential kernels. Training data is plotted as an 'x', pseudo parameters are red circles, and the predictive means are the red dashed lines. The original data is plotted on far left. The next two columns represent draws from the 1-layer and 2-layer models respectively, and the middle and lower plots on the right-most column represent draws from the intermediate layers of the 2-layer model. The single-layer GP can only use one length scale, which it recovers as $l = .461$. Meanwhile, the 2-layer model can adjust for each region, and it composes a layer with length-scale $l = 45.7$ with another with length-scale $l = 3.65$.

scales do not match the originals: we recover $l = 45.7$ for the length-scale in layer 1 compared to $l = 3.65$ for layer 2, resulting in both functions being somewhat spikier than the intermediate region in the original data. However, the predictive draws capture all regions almost perfectly – the outside regions are highly-varying, and the intermediate region is still flat.

The distribution of the pseudo data in the 2-layer model provides insight into how a deep architecture fits the data. In the first layer, the pseudo inputs appear to divide into three clusters, with the intermediate cluster receiving the smallest number of pseudo data points. Thus, it appears the function corresponding to this layer learns that the data comes from three distinct clusters, and that the two outermost regions, which have the highest variance in the data, require the largest number of pseudo data points. Meanwhile, the second layer resembles the original data more

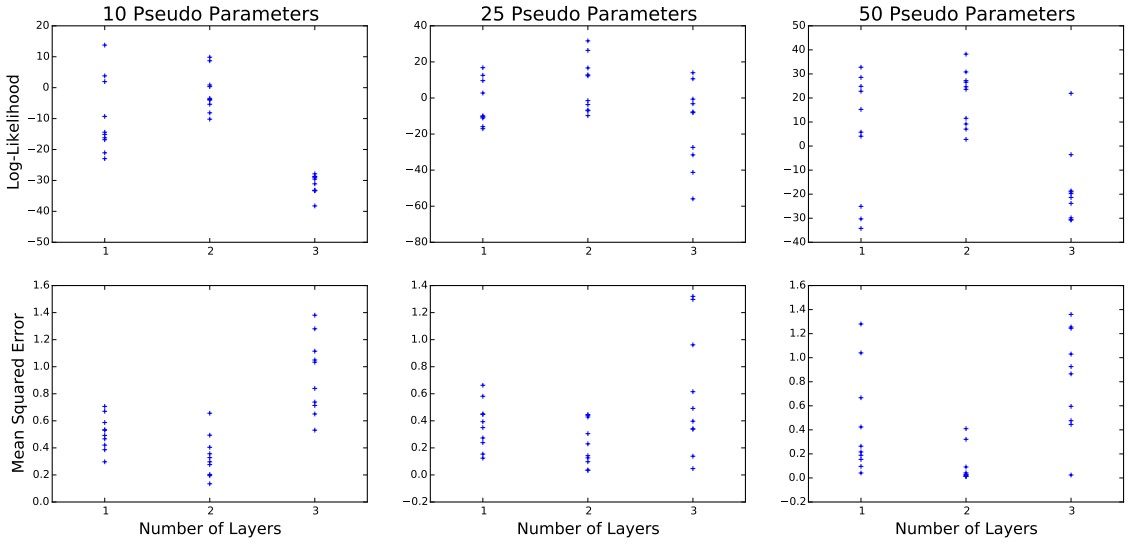

Figure 5.13: Experimental results using the toy non-stationary dataset. For each trial, we use 5 samples to approximate the log marginal likelihood of the model, using single-dimensional hidden layers. We run 10 trials at each combination of the number of layers and size of the pseudo data.

closely, with corresponding spiky and flat regions. Although it is not immediately obvious from the diagrams, the hidden values corresponding to our training examples spread to cover a large range of values, due to the large first-layer length-scale of $l = 45.7$. Thus, the predictive draws corresponding to the second layer are much more spread out. Therefore, while the draws from the second layer GP do not appear to be as sharp as the original data, because the hidden layer extends the input space of the final layer, it can learn the corresponding regions closely.

Furthermore, we run experiments similar to those for the noisy step function. Specifically, we split the data and use 80% for training, making predictions for test data using 1, 2, and 3 layers, while also varying the number of pseudo parameters. We use 5 samples to approximate the log-likelihood and run each combination of parameters 10 times. Our results are depicted in Figure 5.13.

We observe trends similar to those for the noisy step function. The graphs depict a general positive trend in test log-likelihood as we increase from single-layer models to 2-layer models, regardless of the number of pseudo parameters used. Unlike the step function, there do not appear to be any instances where the 2-layer model fails to optimize. Perhaps this is due to using a smaller number of pseudo data points. Here, we use a maximum of 20, while the step function experiment used values 10, 50, and 100. Thus, because we have a smaller number of parameters to fit, our

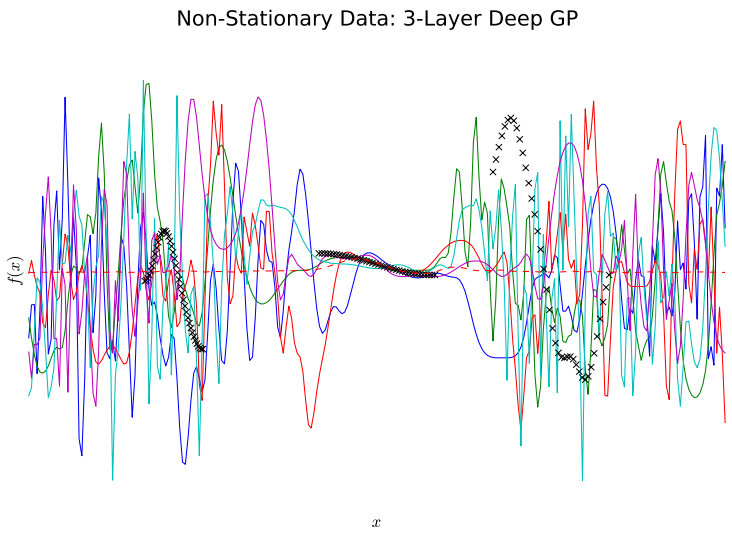

Figure 5.14: Predictive draws from a 3-layer deep GP trained on the toy non-stationary data. We recover length-scales of $l = 16.4$, $l = 2.01$, *and* $l = .128$, *for the first three layers, respectively. The optimization gets stuck in a local optimum, and although the predictive draws are non-stationary, our predictions are poor at the tails.*

optimization routine is less susceptible to local optima.

Interestingly, the 3-layer predictions are consistently worse then those for the 2-layer model, and occasionally worse than those for the single-layer. Again, overfitting does not appear to be an issue – rather, as our architecture scales, it becomes harder to optimize. In Figure 5.14, we plot predictive draws from a 3-layer deep GP fit on the non-stationary data set. Interestingly, the functions are non-stationary, as the middle region is flat and the outer regions are more spiky. However, while it learns values well for the middle region, the same cannot be said for the tails. Judging by the draws in Figure 5.14, there is a much larger predictive variance at the tails, and the predictive mean, calculated by averaging all the sampled means, is flat and non-informative.

5.3 Real World Dataset: Motorcycle

We have demonstrated the success of the DGPS algorithm, along with its shortcomings, for toy data sets. We now explore the algorithm applied to a real data set.

Specifically, we use the motorcycle accident dataset, which is a standard dataset for regression (Silverman, 1985). It consists of 94 points, where the inputs are time in milliseconds since impact of a motorcycle accident and the outputs are the corresponding helmet accelerations. The dataset

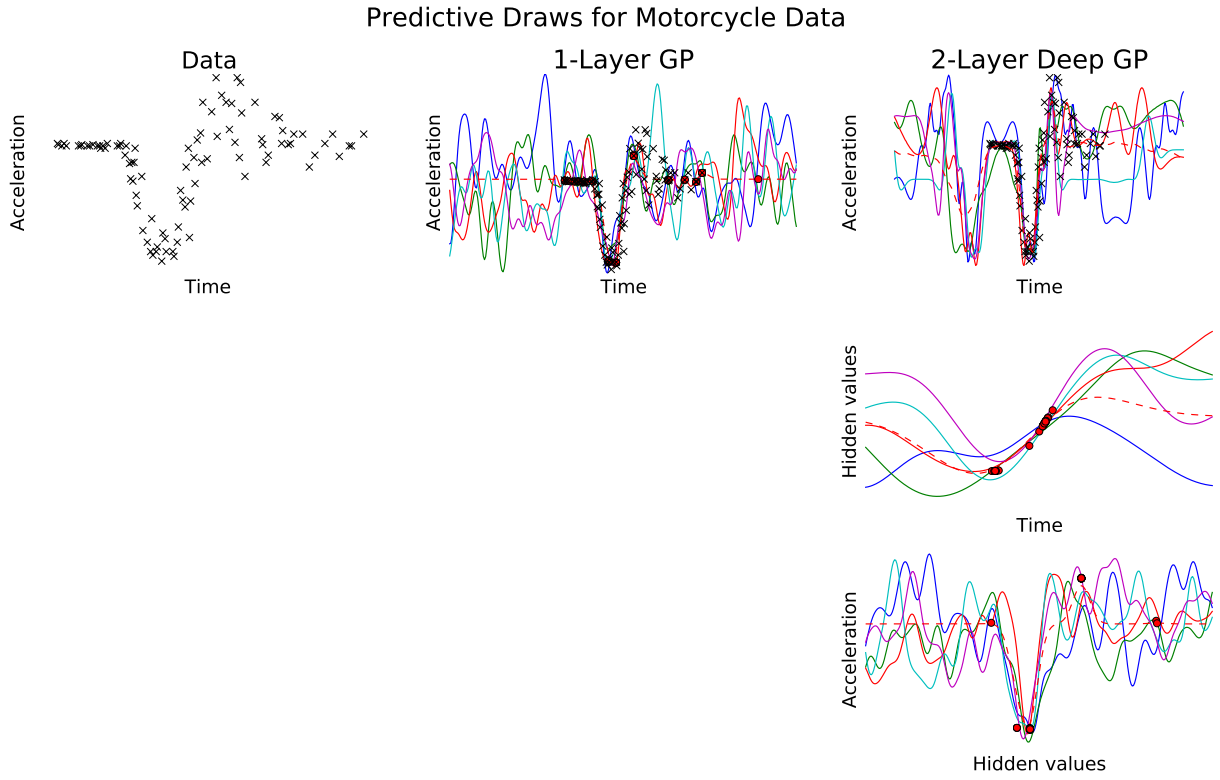

Figure 5.15: Predictive draws from the single-layer and 2-layer models trained on motorcycle data with squared exponential kernels. Original data is on the far left. The single-layer model recovers a length-scale $l = .172$, while the two length-scales in the 2-layer model are $l = 1.61$ and $l = .234$. We use 50 pseudo data points, and sample 10 times at each Monte Carlo step.

is somewhat non-stationary, as the accelerations are constant early on but after a certain time become more varying.

In Figure 5.15, we compare the predictions for a single-layer GP and 2-layer deep GP trained on the motorcycle data. We standardize the inputs and outputs for numerical stability.

The main difference between the models is that the function draws with 2 layers can change shape more dramatically over time. This is partly due to the fact that the 2-layer model consists of the composition of two separate-looking functions: one with a large length-scale, and one with a small length scale. The optimization learns a length-scale $l = 1.61$ in the first layer, followed by a length-scale of $l = .234$ in the second. In composing these length-scales, it can capture the flatness of the left-most region, at the start of the crash, and still recover the spikes as time goes on. In contrast, the single-layer model learns only one length-scale, $l = .172$. As a result, it captures the fluctuations after time zero somewhat well, but cannot flatten out at the beginning. Thus, the

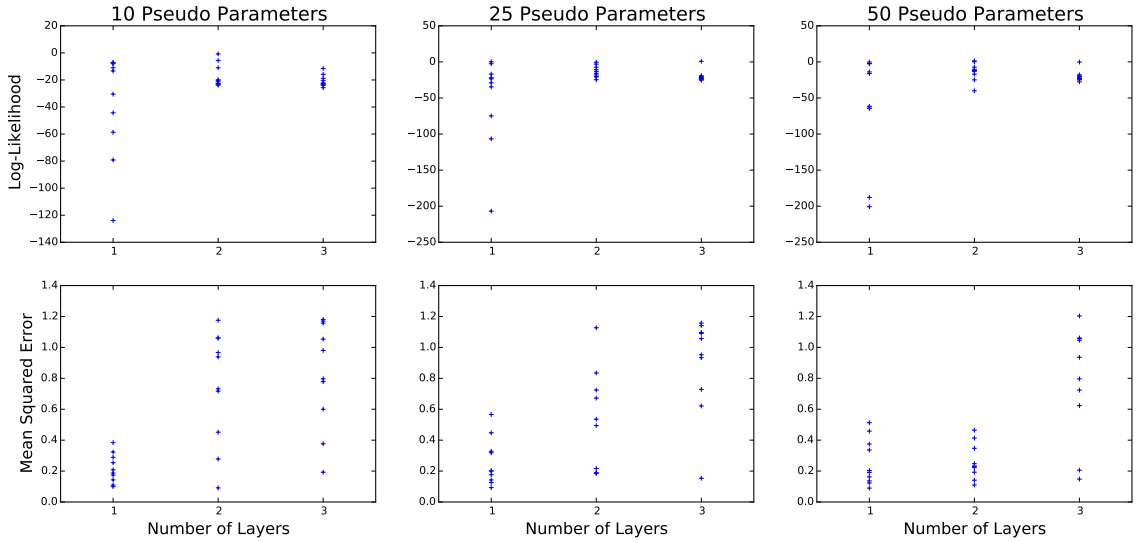

Figure 5.16: Experimental results using the motorcycle dataset. For each trial, we use 5 samples to approximate the log marginal likelihood of the model, using single-dimensional hidden layers. We run 10 trials at each combination of the number of layers and size of the pseudo data.

2-layer model composes functions with different length-scales, one of which is smooth and one of which is more highly-varying, to create a non-stationary set of predictions.

Observing the function, it appears that immediately after the crash, there is not a large degree of acceleration on the biker's helmet. Perhaps this is due to sensitivity in the recording device, or perhaps the crashes do not directly impact the biker's helmet, and it measures force once she falls off her bike. The 2-layer deep GP can draw functions that are flat for this interval yet still capture the spikiness in helmet acceleration immediately afterward. We note that capturing these differences would be possible with a single-layer GP, but it requires the use of a non-stationary kernel or the composition of multiple kernels. With the 2-layer model, we can perform inference without requiring a more sophisticated kernel.

In Figure 5.16, we analyze experimental results on the motorcycle data. We again split the data into an 80/20 train/test split, varying the number of layers between 1, 2, and 3, and varying the number of pseudo parameters between 10, 25, and 50. We use 5 samples to approximate the marginal likelihood at each iteration, having learned from the step function example that 5 samples provides a reasonable approximation without a significant decline in accuracy. Again, we run every trial 10 times.

These results partially match our findings from the step function analysis. The test set log-

likelihood and mean squared errors are similar no matter the size of our pseudo data set, so again it should be encouraged to use less pseudo parameters to ease the computational burden. Looking at the test mean squared error, the range of possible values becomes larger as the number of layers increases. This aligns with the notion that it becomes harder to train deeper architectures. However, the only time the 1-layer deep GP does not outperform its competitors in terms of test MSE is with 10 pseudo parameters. Perhaps this is because, as alluded to before, the additional model parameters make optimization more difficult.

However, test MSE is not the most telling metric to use on our test data. Because we optimize using training likelihood, we should use the same metric to evaluate our test results. Here, there is a definite upward trend in test log-likelihood as the number of layers in our architecture increases. Interestingly, the range of values becomes smaller as the architecture grows, which is the opposite of what we saw with our toy data. It appears that on this particular data set, optimization with one layer is not completely straightforward. Just like the deeper architectures for the step function, we occasionally get stuck in a local optimum, leading to a poor fit and inaccurate predictions.

It is noteworthy, then, that the single-layer case, which has less parameters to fit, is more susceptible to local optima than more complex architectures. There are a couple of possible explanations. For one, we only perform 10 trials at each combination of values, so perhaps this is due to random chance. Additionally, perhaps this real-world data set provides a better example of a non-stationary function than our toy one, and because a single-layer model is not well-equipped to fit non-stationarity, optimization gets stuck more easily. Either way, it is reassuring that deeper architectures can provide better fits and predictions for real-world data sets.

Chapter 6

Conclusion

In this thesis, we have both explored deep Gaussian processes as a model for regression and introduced a novel way to perform inference on a deep GP, which we call the Deep Gaussian Process Sampling algorithm.

We have discussed how the Bayesian nature of single-layer GPs provides estimates of uncertainty and guards against overfitting, while the Fully Independent Training Conditional approximation introduces pseudo data to circumvent computational complexity challenges in inference. Additionally, we discussed the advantages of a deep architecture, especially with regards to learning non-stationary data, along with the disadvantages posed by inference and complexity. To combat these challenges, we proposed the DGPS algorithm, which uses Monte Carlo sampling to approximate the model likelihood and stacks FITC GP units to improve the speed required to train and test.

Finally, we applied the DGPS algorithm to toy and real-world datasets. By doing so, we argued that when deeper models successfully train, they are able to achieve a higher predictive performance than models with less layers. However, it becomes more challenging to optimize our objective as the number of layers increases. None of the models in our experiments suffered from overfitting, demonstrating that even though we explicitly learn all the pseudo parameters, the model still retains its Bayesian regularization advantages.

Moving forward, there are numerous possible extensions to the DGPS algorithm, and many variations to explore. The biggest challenge in inference involves optimizing parameters for deep architectures without getting stuck in local optima. In Chapter 5, we discussed possible methods

to avoid local optima, including trying different optimization routines, such as Adam. Furthermore, smarter initializations and random restarts are possible tweaks that do not involve major algorithmic overhauls.

Because optimization becomes more difficult as we learn more pseudo outputs, we would like to explore models that do not require learning these parameters. We can introduce variational parameters, following the lead of Damianou and Lawrence (2013), requiring us to learn means and variances as opposed to the actual outputs. While this would introduce more parameters, it could provide a closer Bayesian approximation to the model.

Our model performed well on continuous outputs in the context of regression, so it may be worthwhile to explore other types of outputs. A natural extension to the DGPS algorithm is classification. How does our model perform on datasets with binary or discrete outputs?

Finally, the models we have studied in this thesis have featured relatively simple architectures, typically with one dimension for every hidden layer. It would be beneficial to explore the properties of more complex architectures, and to study how a diverse set of hidden units impacts experimental results. Additionally, we can perform model selection on various architectures. By evaluating the model likelihood at different layouts, we can choose the optimal number of hidden layers and units for the particular task at hand.

Bibliography

Bengio, Yoshua and Yann LeCun (2009), "Learning deep architectures for AI." *Foundations and trends in Machine Learning*, 2, 1–127.

Blitzstein, Joseph K. and Carl N. Morris (2016), "Probability for statistical science."

Bui, Thang D, Daniel Hernández-Lobato, Yingzhen Li, José Miguel Hernández-Lobato, and Richard E Turner (2016), "Deep Gaussian processes for regression using approximate expectation propagation." *arXiv preprint arXiv:1602.04133*.

Cybenko, George (1989), "Approximation by superpositions of a sigmoidal function." *Mathematics of Control, Signals and Systems*, 2, 303–314.

Dai, Zhenwen, Andreas Damianou, Javier González, and Neil Lawrence (2015), "Variational auto-encoded deep Gaussian processes." *arXiv preprint arXiv:1511.06455*.

Damianou, Andreas (2015), *Deep Gaussian processes and variational propagation of uncertainty*. Ph.D. thesis, University of Sheffield.

Damianou, Andreas C and Neil D Lawrence (2013), "Deep Gaussian processes." In *Proceedings of the Sixteenth International Workshop on Artificial Intelligence and Statistics (AISTATS-13)*.

Duvenaud, David (2014), *Automatic model construction with Gaussian processes*. Ph.D. thesis, University of Cambridge.

Gill, Philip E and Michael W Leonard (2001), "Reduced-Hessian quasi-Newton methods for unconstrained optimization." *SIAM Journal on Optimization*, 12, 209–237.

Goldberg, Yoav (2015), "A primer on neural network models for natural language processing." *arXiv preprint arXiv:1510.00726*.

Hensman, James and Neil D Lawrence (2014), "Nested variational compression in deep Gaussian processes." *arXiv preprint arXiv:1412.1370*.

Hinton, Geoffrey, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. (2012), "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups." *Signal Processing Magazine*, 29, 82–97.

Houlsby, Neil, Ferenc Huszar, Zoubin Ghahramani, and Jose M Hernández-Lobato (2012), "Collaborative Gaussian processes for preference learning." In *Advances in Neural Information Processing Systems*, 2096–2104.

Hsu, Chih-Wei, Chih-Chung Chang, Chih-Jen Lin, et al. (2003), "A practical guide to support vector classification."

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

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012), "Imagenet classification with deep convolutional neural networks." In *Advances in Neural Information Processing Systems*, 1097–1105.

Livni, Roi, Shai Shalev-Shwartz, and Ohad Shamir (2014), "On the computational efficiency of training neural networks." In *Advances in Neural Information Processing Systems*, 855–863.

Maclaurin, Dougal, David Duvenaud, and Ryan P Adams (2015), "Autograd: effortless gradients in Numpy." *AutoML Workshop*.

Mikolov, Tomas, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur (2010), "Recurrent neural network based language model." In *Interspeech*, volume 2, 3.

Murphy, Kevin P (2012), *Machine Learning: A Probabilistic Perspective*. MIT press.

Neal, Radford M (2012), *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media.

Orbanz, Peter and Yee Whye Teh (2011), "Bayesian nonparametric models." In *Encyclopedia of Machine Learning*, 81–89, Springer.

Quinonero-Candela, Joaquin, Carl Edward Rasmussen, and Christopher KI Williams (2007), "Approximation methods for Gaussian process regression." *Large-scale Kernel Machines*, 203–224.

Rasmussen, C. E. and C. K. I. Williams (2006), *Gaussian Processes for Machine Learning*. The MIT Press.

Shen, Yirong, Andrew Ng, and Matthias Seeger (2006), "Fast Gaussian process regression using kd-trees." In *Proceedings of the 19th Annual Conference on Neural Information Processing Systems*, EPFL-CONF-161316.

Silverman, Bernhard W (1985), "Some aspects of the spline smoothing approach to non-parametric regression curve fitting." *Journal of the Royal Statistical Society. Series B (Methodological)*, 1–52.

Simard, Patrice Y, Dave Steinkraus, and John C Platt (2003), "Best practices for convolutional neural networks applied to visual document analysis." In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, volume 2, 958962, IEEE.

Snelson, Edward and Zoubin Ghahramani (2005), "Sparse Gaussian processes using pseudo-inputs." In *Advances in Neural Information Processing Systems*, 1257–1264.

Yuan, Jin, Liefeng Bo, Kesheng Wang, and Tao Yu (2009), "Adaptive spherical gaussian kernel in sparse bayesian learning framework for nonlinear regression." *Expert Systems with Applications*, 36, 3982–3989.