Scalable Gaussian Processes for Economic Models

Thomas Michaelsen Pethick

Department of Applied Mathematics and Computer Science
The Technical University of Denmark

This dissertation is submitted for the degree of
Master of Science in Computer Science and Engineering

August 2019

Acknowledgements

First of all I would like to thank my supervisor at EPFL, **Volkan Cevher**, for welcoming me into his lab and really making me feel at home. It was inspiring to experience how a lab could be run with both compassion and determination. For the people in the lab I would especially like to thank **Paul Rolland** for the endless discussions at the whiteboard of which I hope many more will come—it was always a joy.

I would also like to thank **Simon Scheidegger** for being so helpful and patient in introducing me to the economic setting. He always followed up with detailed responses, even when occupied with running a workshop in Chicago, and I truly appreciated that.

I also want to thank my supervisor at my home university, **Ole Winther**, for letting me pursue my interests and for useful pointers when needed.

Finally, I would like to thank **Jacob R. Gardner** and **Geoff Pleiss** for being so responsive with my inquiries about in particular Blackbox Matrix-Matrix GPs. More generally, I am grateful to **the entire open source community** which I find is too rarely acknowledged. Without it there would have been no foundation on which to build my thesis. The same can be said about **the entire academic community** behind, in particular, Gaussian Processes. More than anyone, they have shaped how I write and think. Thank you.

Abstract

In economics it is common to analyse consumers' and companies' behavior by solving a stochastic optimal control problem. In order to solve these problems we need to model the so-called value and policy function in some way. Recently these methods have been scaled to up to an astonishing 500 dimensions. This has been made possible by two state-of-the-art methods: Adaptive Sparse Grids and Active Subspaces. Where the former is incapable of dealing with noisy data the latter is limited by its cubic computation cost and strong assumptions on the function.

It is of interest to assess the uncertainty in the policy function. To this end we explore Gaussian Processes (GPs) which provide well-calibrated confidence intervals. There remain three big challenges for GPs in our setting: non-stationarity common to economics problems, the high-dimensionality and handling many observations.

To handle non-stationarity and high-dimensionality we study Deep Kernel Learning (DKL), which we compared to both GPs and neural network counterparts with and without a Bayesian linear regressor. We show empirically that DKL is only interesting to consider when either the assumptions made by the earlier state-of-the-art models break down or the computational cost becomes prohibitive.

For tackling scalability we focus on inducing point methods and a generalization of them called KISS-GP. Combined with DKL it is possible to apply KISS-GP even to high-dimensional problems. As a proof-of-concept we apply this approach to a stochastic optimal growth model suggesting that the method could scale to unprecedented dimensionality greater than 500.

In this thesis we lay the groundwork for applying these expressive GP models in an economics setting. A significant part of the contribution consists of the code base which permits systematic testing of the models on various functions and data sets. With this code base several models can be run in parallel on a High Performance Computing cluster, aggregated and viewed—all through a Jupyter Notebook. We hope this will accelerate iteration cycles and reproducibility for future investigations.

Table of contents

List of figures **xi**

List of tables **xiii**

1 Introduction **1**
1.1 Prior work . . . 2
1.2 Contributions . . . 3
1.3 Outline . . . 3
1.4 Audience . . . 4

2 Background **5**
2.1 Stochastic Optimal Growth Model . . . 5
2.2 Gaussian Processes for Regression . . . 6
2.2.1 Gaussian Process definition . . . 6
2.2.2 Gaussian Process Regression . . . 9
2.3 Scalability . . . 15
2.3.1 Low Rank Approximation . . . 16
2.3.2 Subset of Regressors . . . 18
2.3.3 Variational Sparse Gaussian Processes . . . 19
2.3.4 Kernel Interpolation for Scalable Structured Gaussian Processes 20
2.3.5 Sparse Spectrum Gaussian Processes . . . 23
2.4 High-dimensionality . . . 25
2.4.1 Deep Kernel Learning . . . 27
2.4.2 Scalable Deep Kernel Learning . . . 28
2.4.3 Active Subspaces . . . 29
2.5 Non-stationarity . . . 31
2.5.1 Deep Kernel Learning . . . 31
2.6 A Unified View of the Models . . . 32

3 Experiments **35**
3.1 Setup . . . 35
3.1.1 Methods . . . 35
3.1.2 Error Metrics . . . 37
3.2 Model Configuration . . . 37
3.2.1 Expressiveness of Deep Kernel Learning . . . 37
3.2.2 Neural Network Architecture . . . 39
3.2.3 Greedy Training . . . 41
3.3 High Dimensional Embeddings . . . 42
3.3.1 Linear Embeddings . . . 42
3.3.2 Non-linear Embeddings . . . 46
3.4 Comparison with Many Datapoints . . . 49
3.4.1 Influence of the Number of Inducing points . . . 49
3.4.2 Improving by increasing number of observations . . . 52
3.5 Economic Setting . . . 54
3.5.1 Heston Option Pricing . . . 54
3.5.2 Stochastic Optimal Growth Model . . . 57
3.5.3 High-dimensional Policy Function . . . 58

4 Implementation **63**
4.1 Settings and Configurations . . . 63
4.1.1 The power of configurations . . . 64
4.2 Model execution . . . 64
4.2.1 Server Execution . . . 65
4.2.2 Local execution . . . 66
4.2.3 Execution Variants . . . 66
4.3 Modules . . . 66
4.3.1 Models . . . 67
4.3.2 Environments . . . 67
4.3.3 Configuration . . . 68
4.3.4 Context . . . 69
4.4 Conclusing remarks . . . 69

5 Discussion **71**
5.1 High-dimensionality . . . 71
5.2 Scalability . . . 72
5.3 Uncertainty Quantification . . . 73

5.4 Economic settings . . . 73
5.5 Conclusion . . . 74

References **75**

Appendix A Theory **79**
A.1 Linear Algebra . . . 79
A.1.1 Woodbury, Sherman and Morrison Formula . . . 79
A.1.2 Sylvester's Determinant Identity . . . 79
A.1.3 Inverse of a partitioned matrix . . . 79
A.1.4 Marginalization of product . . . 80
A.1.5 Norms . . . 80

Appendix B Practical Aspects **81**
B.1 Initialization . . . 81
B.2 Normalization . . . 81
B.3 Normalization of the feature space . . . 83
B.4 Hyperparameter Optimization . . . 83
B.5 Loose Stopping Critierion during Training . . . 84
B.6 Trouble-shooting Low Noise . . . 85

Appendix C Experiments **89**
C.0.1 Embeddings . . . 89
C.0.2 High-dimensional Policy Function . . . 89
C.0.3 Kernel reconstruction . . . 89

List of figures

2.1 Function samples for three different length-scales 10
2.2 Posterior distribution for three different kernels 10
2.3 Active Subspace on 2D toy example 30

3.1 Varying length-scale modelled with DKL 38
3.2 2D embedding modelled by DKL 39
3.3 Activation functions effect on posterior 40
3.4 Linear exponential synthetic test function 43
3.5 Hyperparameter optimization for embeddings 46
3.6 Linear embedding model comparison 47
3.7 Linear embedding predictions in feature space 47
3.8 Failure case for Active Subspaces 48
3.9 Natural sound modelling with increasing M 50
3.10 SSGP variance starvation 52
3.11 Natural sound modelling with increasing N 53
3.12 Heston option pricing 55
3.13 Value function convergence for the optimal growth model 59
3.14 2-dimensional policy function 60

4.1 Server execution flow 66

B.1 Initial hyperparameters are important 82
B.2 Numerical instability in low-noise 87

C.1 Kernel approximation 91

List of tables

2.1 A unified view of the models 33
2.2 Time and space complexity for models 33

3.1 Results for non-linear embedding of Sinc 48
3.2 Heston based option pricing results 56
3.3 2-20 dimensional discontinuous policy functions 61

4.1 Error metrics 65
4.2 Implemented models 67

C.1 Linear embeddings 90
C.2 MNLP for 2-20 dimensional discontinuous policy functions 91

Chapter 1

Introduction

To analyse the behavior of economic agents such as consumers over time the Dynamic Stochastic General Equilibrium model has become a fundamental tool in macroeconomics. The model makes several assumptions about the consumers: they live forever and try to optimize for happiness, which depends on consumption and leisure restricted by some budget. In addition they are inherently impatient, so they discount future happiness by some constant factor. In this setting the goal is to find the optimal behavior for each agent given this measure of happiness. The problem is usually formulated in such a way that there is an optimal steady state. That is, a state which no optimal agent would leave—so it suffices to find a solution for a single state for which there will be a constant policy (by which we mean behavior).

Whether this steady state exists depends on the assumptions, however. Imagine that we introduced stochastic shocks that could alter the state in the economy. Then the behavior of the agents would vary. So instead of finding a solution at one state we wish to find a *global solution*. In other words, our interest is to find a good approximation of the policy at every possible state. In general there is no analytical solution so we have to use approximate methods.

When solving such a problem we need to interpolate between a discrete set of known policies on the state space. Several interesting challenges arise for this particular regression problem. First of all, the large stochastic shocks we introduced lead to highly non-linear policies with discontinuities. Secondly, it is of interest to scale the problem to include several agents such as industrial organization. This grows the dimensionality of the problem both in terms of actions and states. Further, both the discontinuities and the high-dimensionality makes it a statistically challenging problem. So our regression method of choice will ultimately have to scale in terms of observations.

Since we are now *estimating* the policy it is natural to attempt to quantify the uncertainty arising from the interpolation. There are several other sources of uncertainty such as algorithmic uncertainty and parameter uncertainty which would be interesting regardless.[1] This is because we are interested in inspecting the solution we find to understand the *model* better. The actual applications of this uncertainty is outside the scope of this thesis however.

Gaussian Processes (GPs) provide an elegant non-parametric solution to quantifying the uncertainty on a continuous response surface. We still face several of the previously mentioned challenges as it is not geared towards the application otherwise. That is, GPs popularity has primarily been driven by the use of simple covariance functions such as the squared exponential kernel. These kernels incorporate very strong smoothness assumptions which makes them ill-fitted for modelling discontinuous functions. Further, they rely on the euclidean distance which becomes uninformative in high-dimensions. Lastly, performing exact GP regression scales cubically in the number of observations making it prohibitively expensive for much more than 10^4 data points. We aim to investigate these three aspects and test them in the financial setting.

1.1 Prior work

Injecting regression models into algorithms for solving these economic models have already been well studied. Particularly for tackling high dimensional economic problems, two recent state-of-the-art approaches are that of Adaptive Sparse Grids (A-SG) [Brumm and Scheidegger] and Active Subspaces with gaussian processes (AS-GP) [Scheidegger and Bilionis]. However, they have their limitations.

A-SG cannot deal with noisy data in a principled way since it interpolates linearly between observations. This is problematic in most realistic settings where some level of noise is unavoidable. Secondly, those observations have to be placed on a grid. Even if we control the observation locations as in the value iteration (introduced in section 2.1) this can still provide an obstacle. This is because the simulation run to obtain that observation might not complete in a reasonable time-frame. If it is not possible to ignore an observation so our algorithm can only run as fast as our worst case simulation.

AS-GP was introduced to deal with exactly these problems but has its own difficulties. Most noticeably AS are restricted to function for which we can obtain a gradient. This is not too much of a restriction since finite difference approximations can be used

[1] For the curious reader we refer to [Smith].

instead. More problematic is its strong assumption of a *linear* manifold. We cannot always hope to have such a simple embedding which is why we will consider more expressive models. Furthermore, the computational complexity of $\mathbb{O}(N^3)$ associated with both AS and GP can become a restriction in very high dimensions where many observations are needed to learn the subspace.

1.2 Contributions

We have so far provided one motivational example for applying GPs in an economics setting but the range of problems is larger. The aim is more broadly to provide preliminary work for a new department taking on researching the application of GPs to financial and economic problems at EPFL[2].

The contribution of this thesis has three components:

- **A literature overview** that covers the three aspects of concern in the context of GPs: high-dimensionality, non-stationarity and scalability.
- **A code base**[3] for conducting experiments using a selection of models on a variety of both synthesized test-functions and real world data sets. The workflow allows for running reproducible experiment in a High Performance Computing (HPC) environment and collecting them for aggregation and inspection afterwards.
- **A proof-of-concept** which more closely studies an approach that treats the dimensionality by assuming a low dimensional manifold and the scalability by inducing points on a grid (see section 2.4.1 and section 2.3.4 respectively). We focus our study on embedded functions in higher dimensions than in the original paper and include a comparison with AS-GP. This model is then applied to various economic problem. Of particular interested is the injection into a value iteration scheme to solve a dynamic optimal control problem.

1.3 Outline

Chapter 2 will provide the necessary theoretical background by first introducing the motivational example in the economic setting together with the Value iteration algorithm as one possible solution. GPs will then be thoroughly reviewed as a regression

[2]This is a collaboration between the department of Volkan Cevher and Simon Scheidegger.
[3]The code base is freely available at `https://github.com/tmpethick/thesis-code`.

model that can be injected into this iterative scheme. We will do so by treating the three main challenges for GPs separately: scalability, high-dimensionality, and non-stationarity.

In chapter 3 we study the models introduced in the previous theory chapter. We conduct several experiments to understand the expressive Deep Kernel Learning method better and carry out a comparative analysis of the various scalable methods. Finally we consider several financial and economic application as case studies. The implementation that made these experiments possible are described in the subsequent chapter 4 which concerns itself with the practical contributions.

The discussion in chapter 5 comments on which methods are promising and in which directions to develop them. It further considers other methods to attack the same problems.

1.4 Audience

This thesis is written partially as a guide book for practitioners working with financial data. To this end, so called *recommendation* sections are to be found throughout the sections which will provide practical advice. In general we will try to be as precise about specific parameter settings and running times when possible.

However, we will still strive at being theoretically grounded but hopefully manage to give an intuition for GPs. When developing more intricate approaches such as KISS-GP we will do so by connecting it to simpler ones to make them more accessible. Even though we will be fairly self-contained in building up the theory we will assume familiarity with basic linear algebra, probability and notions such as convexity. A basic knowledge of machine learning with such concepts as training loss, back-propagation, supervised and unsupervised loss (i.e. mean square error and reconstruction error respectively) is also expected.

Chapter 2

Background

2.1 Stochastic Optimal Growth Model

This economic model taken from [Scheidegger and Bilionis] considers D sectors that at every time t have a capital stock associated with it summarized by the *state vector* $\boldsymbol{k}_t = (k_{t,1}, \ldots, k_{t,D})$. At every discrete time-step the sectors chooses their consumption, investment level and so called elastic labor supply level denoted $\boldsymbol{l}_t$, $\boldsymbol{C}_t$ and $\boldsymbol{I}_t$ respectively subject to some constraints. These *control variables* influence the state vector over time. We are then interested in optimizing some *utility* defined on $\boldsymbol{c}_t$ and $\boldsymbol{l}_t$ in expectation.

This is an infinite-horizon stochastic optimal control problem that we can describe more generally: At every time t we have a state vector $\boldsymbol{x}_t$. We obtain a control variable $\boldsymbol{\xi} \in \Xi$ through a time-invariant *policy function* $p : X \to \Xi$ that is fully defined given the state. Mechanically the state then evolves to the next period through a distribution depending *only* on the current state and policy

$$\boldsymbol{x}_{t+1} \sim F\left(\cdot | \boldsymbol{x}_t, p\left(\boldsymbol{x}_t\right)\right), \tag{2.1}$$

F is assumed known so the objective is to find the policy which maximizes the *value function*

$$V^*\left(\boldsymbol{x}_0\right) = \max_{\{\boldsymbol{\xi}_t\}_{t=0}^{\infty}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(\boldsymbol{x}_t, \boldsymbol{\xi}_t\right) \tag{2.2}$$

where the expectation is taken over the distribution of $\boldsymbol{x}_t$, $\beta \in [0, 1]$ is the discount factor, $\boldsymbol{x}_{t+1} \sim F\left(\cdot | \boldsymbol{x}_t, \xi_t\right)$ and u is our *utility function*. Notice how tricky this problem is since its solution is the infinite sequence of controls that maximizes eq. (2.2). Fortunately *value iteration* comes to our rescue.

Value iteration The *principle of optimality* [Bellman] says that whatever state and decision we make at time t, the policy should be optimal onwards. This allows us to break this problem into smaller sub-problems in which we solve for the optimal policy at time $t-1$ and then, given this solution, recursively solve at t. Notice that we are in an infinite-horizon setting and recurse backwards. So some initial guess for $V_\infty(\cdot)$ is needed after which we can proceed in the following fashion:

$$V_i(\boldsymbol{x}) = \max_{\boldsymbol{\xi}} \left\{ u(\boldsymbol{x}, \boldsymbol{\xi}) + \beta \mathbb{E} V_{i+1} \left[\boldsymbol{x}_{t+1} \right] \right\}. \tag{2.3}$$

where $\boldsymbol{x}_{t+1} \sim F(\cdot|\boldsymbol{x}, \boldsymbol{\xi})$ over which the expectation is taken. Under certain technical conditions this sequence of value functions will converge to $V^*(\cdot)$.

This is still difficult to solve as we have to maximize over the optimal control *for every $\boldsymbol{x}$ in the continuum.* In practice we evaluate $V_i(\cdot)$ at a discrete set of points so some model is required to interpolate between these points. An overview of the algorithm is given in algorithm 1.

This high-level description will suffice but the theory behind these optimal control problems is rich and has a lot more subtleties to it. For a more complete description of the optimal growth model, in particular, we refer to [Scheidegger and Bilionis].

Procedure 1 Value iteration for continuous state spaces.

Input: Utility u, transition density F, model $\mathcal{M}$
Output: $V(\cdot)$
  $V(\cdot) \leftarrow$ initial guess
  repeat
    for all $\boldsymbol{x} \in \{ \boldsymbol{x}_i \}_{i=1}^N$ **do**
      $y_i \leftarrow$ compute eq. (2.3) using previous $V(\cdot)$
    end for
    $V(\cdot) \leftarrow$ build model $\mathcal{M}$ on $\{ (\boldsymbol{x}_i, y_i) \}_{i=1}^N$
  until $V(\cdot)$ converges.

2.2 Gaussian Processes for Regression

To describe Gaussian Processes we will mostly be following the work of [Rasmussen and Williams].

2.2.1 Gaussian Process definition

Gaussian processes (GPs) can be seen as defining a probability distribution over functions. This view provides an easy intuition and is illustrated in section 2.2.1.1

where different sample functions are drawn from a GP prior. However, to develop GPs we will take a slightly different point of view and instead consider a collection of random variables $f(\boldsymbol{x})$ indexed by the continuos domain $\mathcal{X}$. Suppose we then pick a finite subset of these random function values $\boldsymbol{f} = \{f_1, f_2, \ldots, f_N\}$ at corresponding indexes $\mathbf{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$. Then a GP is simply defined as follows.

Definition 2.2.1 (Gaussian Process)**.** A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution [Rasmussen and Williams, p. 13]:

$$p(\boldsymbol{f}|\mathbf{X}) = \mathcal{N}(\boldsymbol{\mu}, \mathbf{K}).$$

Since a Gaussian distribution is fully defined by its mean and covariance matrix we need a way to specify this for *any* finite number of random variables on our domain. To this end we define a mean function $m(\boldsymbol{x})$ and covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ such that for a finite number of random variables the joint Gaussian distribution is defined as,

$$\mathcal{N}(\boldsymbol{\mu}, \mathbf{K}) \quad \text{where} \quad \begin{aligned} \boldsymbol{\mu}_i &= m(\boldsymbol{x}_i), \\ \mathbf{K}_{ij} &= k(\boldsymbol{x}_i, \boldsymbol{x}_j). \end{aligned} \tag{2.4}$$

From here-on we will only consider GP priors with zero mean. This might seem like a restriction for the regression task but in practice we can work around it by simple data augmentation such as normalization. This leaves the covariance function to fully define our prior.

2.2.1.1 Covariance function

The covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ (also referred to as the *kernel*) captures the correlation between two points which is expressed as follows, assuming zero mean,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}\left[f(\boldsymbol{x}) f(\boldsymbol{x}')\right]. \tag{2.5}$$

Notice that it thus expresses the similarity between *function values* even though k is determined by the input $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}$.

The only requirement we have in k is that it induces a valid covariance matrix (i.e. the matrix is symmetric and positive definite $\mathbf{v}^\top \mathbf{K} \mathbf{v} \geq 0, \forall \mathbf{v}$). We will cover a few of the most relevant families of covariance functions relevant for this thesis. For a comprehensive overview of classical kernels we refer to [Rasmussen and Williams].

Squared Exponential For many problems it is reasonable to assume that points which are close have similar output as well. Typically this is captured with a covariance functions which decays smoothly with distance. One such example is the popular Squared Exponential (SE) covariance function

$$k_{\text{SE}}(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2 \exp(-\frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{2\lambda^2}). \tag{2.6}$$

Notice that it only depends on the *closeness* of points since it is written as a function of $\boldsymbol{x} - \boldsymbol{x}'$. We call this family of translation-invariant kernels *stationary* kernels which will later prove to be an important property for scaling GPs.

The SE kernel has two so-called *hyperparameters* which control the properties of the sample functions. The *output variance* σ^2 determines the amplitude while λ is the typical *length-scale* over which the function will vary. Collectively we will refer to them as $\boldsymbol{\theta}$ which will be relevant when we consider optimizing over them in section 2.2.2.3.

Figure 2.1 depicts sample-draws from SE kernels of three distinct length-scales. In practice these samples are created by finely discretizing the domain and then constructing the covariance matrix using the covariance function. A draw is then simply a sample from a multivariate gaussian using that covariance matrix. Notice how the samples stay around the zero mean prior with a wiggliness inversely proportional to the length-scale hyperparameter.

As can be seen in fig. 2.1 the sample draws are very smooth. In fact, functions sampled from a SE are infinitely differentiable. This is a rather strong assumption which might not fit realistic regression tasks. For this reason a bigger family of stationary kernels can be considered which includes the square exponential kernel, namely the Matérn kernel.

Matérn The stationary kernel has the following form:

$$k_{\text{Matérn}}(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}(\boldsymbol{x} - \boldsymbol{x}')}{\lambda} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}(\boldsymbol{x} - \boldsymbol{x}')}{\lambda} \right), \tag{2.7}$$

where ν and λ are positive and K_ν is a modified Bessel function of the second kind. The new hyperparameter ν determines how smooth the function is: it is k-times differentiable if and only if $k < \nu$. So not surprisingly we obtain the infinitely differentiable SE kernel when $\nu \to \infty$. If we want *rougher* priors we pick a smaller ν. In practice it is especially convenient to pick $\mu = p + 1/2$ as the Matérn simplifies to a product of an

exponential and polynomial of order p. In particular for $\nu = 3/2$:

$$k_{Matérn3/2}(\boldsymbol{x}, \boldsymbol{x}') = \left(1 + \frac{\sqrt{3}(\boldsymbol{x} - \boldsymbol{x}')}{\lambda}\right) \exp\left(-\frac{\sqrt{3}(\boldsymbol{x} - \boldsymbol{x}')}{\lambda}\right) \tag{2.8}$$

which is only once differentiable. This will provide a fairer baseline than that of the SE kernel when considering using GPs for non-stationary functions.

For finance it is of interest to notice the connection with the Ornstein-Uhlenbeck Process as we obtain the corresponding kernel, the exponential covariance function $k(r) = \exp(-r/\lambda)$, when $\mu = 1/2$. So this family even includes kernels that are not differentiable.

Linear The linear kernel produces samples that are linear functions

$$k_{\text{Linear}}(\mathbf{x}, \mathbf{x}') = \sigma^2 \mathbf{x}\mathbf{x}'^{\top} \tag{2.9}$$

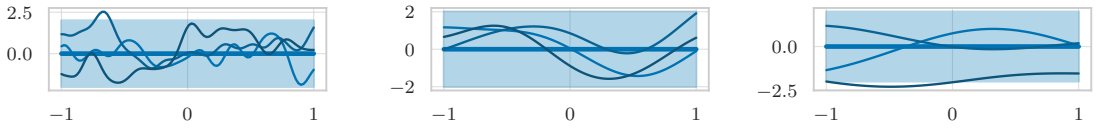
This kernel family is *not* stationary—as a simple counterexample consider $\boldsymbol{x} = \boldsymbol{x}'$ and notice that the kernel grows as a function of positive $\boldsymbol{x}$. The *variance* hyperparameter σ^2 controls how quickly this increase is away from the origin.

It is an interesting special case of kernels as a GP with a linear kernel has been shown to be equivalent to a Bayesian Linear Regressor. We will later exploit this to gain scalable approaches.

A Note on Notation The kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$ and the covariance matrix $\mathbf{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ that it induces on a collection of inputs $\{ x_i \}_{i=1}^N$ will be used interchangeable. To distinguish between covariance matrices constructed from different collections we will define $\mathbf{K}_{NN}$ as being constructed from the collection using the symbol N for its number of symbols. This will be especially useful when we talk about inducing points methods in section 2.3.1.1.

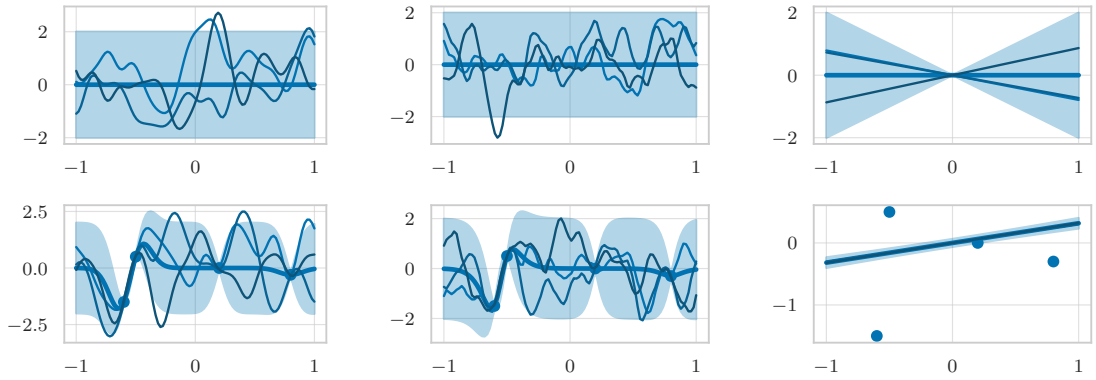
2.2.2 Gaussian Process Regression

Up until now we have seen how different covariance function leads to different sampled functions and that these are influenced by the hyperparameters of the covariance function. We are ultimately interested in conditioning on the observed data to make predictions. To this end we will use the GP as our modelling prior and through Bayesian inference derive the posterior used for predictions.



(a) $l = 0.1$ (b) $l = 0.5$ (c) $l = 1.0$

Fig. 2.1 Sample draws from a squared exponential kernel with length-scale 0.1, 0.5 and 1.0 from left to right. Each plot shows the mean and 95% confidence interval as well as three sample draws.



(a) $k_{\text{SE}}(\boldsymbol{x})$ (b) $k_{\text{Matérn5/2}}(\boldsymbol{x})$ (c) $k_{\text{Linear}}(\boldsymbol{x})$

Fig. 2.2 Three different kernels each showing the prior with mean zero (top) and the posterior after conditioning on four points (bottom). Each plot shows the mean and 95% confidence interval as well as three sample draws.

2.2.2.1 Noise-free setting

We first consider the noise-free setting in which we are given a training dataset $(\boldsymbol{f}, \mathbf{X}) = (\{ f_i \}, \{ \boldsymbol{x}_i \})_{i=1}^n$ of n inputs $\boldsymbol{x}_i$ and associated outputs $f_i = f(\boldsymbol{x}_i)$ and asked to compute the predictive distributions $\boldsymbol{f}_T = \{f_t\}_{t=1}^T$ at $\mathbf{X}_T = \{\boldsymbol{x}_t\}_{t=1}^T$ (also referred to as the test set).

To make a predictions first observe that our $\boldsymbol{f}$ and $\boldsymbol{f}_T$ are jointly gaussian distributed under our GP prior

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f}_T \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K}_N & \mathbf{K}_{NT} \\ \mathbf{K}_{TN} & \mathbf{K}_T \end{bmatrix}\right), \tag{2.10}$$

where we defined $\mathbf{K}_{AB}$ as the covariance matrix formed by $\mathbf{K}_{ij} = k(\boldsymbol{X}_i, \boldsymbol{Y}_j)$ where $\mathbf{X}$ and $\mathbf{Y}$ are the input location sets of length A and B respectively. This block structure will be convenient when we now write down the posterior.

One of the nice properties of GPs is that we can now derive an analytical expression for the posterior $\boldsymbol{y_T}|\mathbf{X}_T, \mathbf{X}, \boldsymbol{y}$ at the desired predictive locations given our training data. It only requires a bit of manipulation of the joint gaussian distribution expressed above to arrive at (see section A.1.4)

$$\boldsymbol{y_T}|\mathbf{X}_T, \mathbf{X}, \boldsymbol{y} \sim \mathcal{N}\left(\mathbf{K}_{TN}\mathbf{K}_{NN}^{-1}\boldsymbol{f}, \mathbf{K}_{TT} - \mathbf{K}_{TN}\mathbf{K}_{NN}^{-1}\mathbf{K}_{NT}\right). \tag{2.11}$$

In practice we will mostly have to deal with noisy settings which will be covered next. However the computational complexity for both settings are the same. So when we analyse more intricate schemes such as KISS-GP in section 2.3.4 we will do so in this simpler setting.

2.2.2.2 Noisy setting

We have so far considered modeling $\boldsymbol{f}$ and $\mathbf{X}$. However, in many applications we do not have direct access to $\boldsymbol{f}$ but observed some noisy version of it. We will consider the simplest case of homogeneous additive Gaussian noise such that the true (deterministic) function $f(\boldsymbol{x})$ is only observed through y given by

$$y_i = f(\boldsymbol{x}_i) + \boldsymbol{\varepsilon}_i \quad \text{where} \quad \varepsilon_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}\left(0, \sigma^2_{\text{noise}}\right)$$

such that our training and test variables now include $\boldsymbol{y}$ and $\boldsymbol{y}_T$ respectively. The task is now to predict $\boldsymbol{y}_T$ at $\mathbf{X}_T$ after having observed $\boldsymbol{f}$ at $\mathbf{X}$.

We can equivalently write our noise model as

$$p(\boldsymbol{y}|\boldsymbol{f}) = \mathcal{N}\left(\boldsymbol{f}, \sigma^2_{\text{noise}}\,\mathbf{I}\right). \tag{2.12}$$

If we integrate out the latent $\boldsymbol{f}$ where $p(\boldsymbol{f}|\mathbf{X}) = \mathcal{N}(\mathbf{0}, \mathbf{K})$ (see section A.1.4) we obtain

$$p(\boldsymbol{y}|\mathbf{X}) = \int p(\boldsymbol{y}|\boldsymbol{f})p(\boldsymbol{f}|\mathbf{X})d\boldsymbol{f} = \mathcal{N}\left(\mathbf{0}, \mathbf{K} + \sigma^2_{\text{noise}}\,\mathbf{I}\right) \tag{2.13}$$

Consider the joint distribution of both noisy test and training variables

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y}_T \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K}_N & \mathbf{K}_{NT} \\ \mathbf{K}_{TN} & \mathbf{K}_T \end{bmatrix} + \sigma^2_{\text{noise}}\mathbf{I}\right). \tag{2.14}$$

The predictive distribution then follows in a similar fashion to eq. (2.11) in the noise-free setting,

$$\begin{aligned} p\left(\boldsymbol{y}_T|\mathbf{X}_T\mathbf{X}\boldsymbol{y}\right) = \mathcal{N}\Big(&\mathbf{K}_{TN}\left[\mathbf{K}_N + \sigma^2_{\text{noise}}\mathbf{I}\right]^{-1}\boldsymbol{y}, \\ &\mathbf{K}_T - \mathbf{K}_{TN}\left[\mathbf{K}_N + \sigma^2_{\text{noise}}\mathbf{I}\right]^{-1}\mathbf{K}_{NT} + \sigma^2_{\text{noise}}\mathbf{I}\Big). \end{aligned} \tag{2.15}$$

The computational bottleneck is in solving the linear system involving the inverse of the kernel. In practice this is commonly done with Cholesky decomposition $\mathbf{K} = \mathbf{L}\mathbf{L}^\top$ for numerical stability (see section 2.2.2.5). In additional we will briefly cover the alternative method of conjugate gradients to solve the linear system in section 2.2.2.6. It has become popular recently because it has made possible highly parallelizable code suitable for GPU's. However, first we will turn to the other computational demanding part of GPs—that of optimizing the kernel hyperparameters.

2.2.2.3 Model Selection

We saw in fig. 2.1 how the length-scale heavily influences the prior the SE kernel places on our functions. These hyperparameters lead to a collection of possible models of which we would like to pick the one that best describe our data. Ideally we would be fully bayesian and find the distribution over $\boldsymbol{\theta}$ given the data and some prior over $\boldsymbol{\theta}$

$$p(\boldsymbol{\theta}|\boldsymbol{y}, \mathbf{X}) = \frac{p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\boldsymbol{y}|\mathbf{X})}. \tag{2.16}$$

The normalization constant $p(\boldsymbol{y}|\mathbf{X})$ can be obtained by marginalizing out $\boldsymbol{\theta}$ but depending on the form of $p(\boldsymbol{\theta})$ this might not be analytically tractable and we would have to resort to sampling approaches like Markov Chain Monte Carlo.

A simpler solution is to approximate with a point estimate by maximizing the hyperparameter distribution (noticing that the problematic normalizing factor disappears)

$$\boldsymbol{\theta}^{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\boldsymbol{y}, \mathbf{X}) = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{2.17}$$

This is commonly referred to as the maximum a posteriori (MAP). Picking a uniform prior we obtain a particularly simple form

$$\boldsymbol{\theta}^{ML} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}), \tag{2.18}$$

where the $p(\boldsymbol{y}|\mathbf{X})$ is the *marginal likelihood* (ML) which name become evident when noticing that we in practice compute it by marginalizing the latent function

$$p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\boldsymbol{y}|\boldsymbol{f})p(\boldsymbol{f}|\mathbf{X}, \boldsymbol{\theta})d\boldsymbol{f}. \tag{2.19}$$

To derive the analytical form, notice that $p(\boldsymbol{f}|\mathbf{X}, \boldsymbol{\theta})$ is distributed as our prior $\mathcal{N}(0, \mathbf{K}_{\boldsymbol{\theta}})$. We can then apply section A.1.4 to derive it in closed form. For numerical stability we also take the logarithm which splits the expression into a sum of particularly *two* interesting terms

$$\log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = \underbrace{\boldsymbol{y}^\top \left(\mathbf{K}_{\boldsymbol{\theta}} + \sigma^2_{\text{noise}}\mathbf{I}\right)^{-1} \boldsymbol{y}}_{\text{model fit}} + \underbrace{\log \left|\mathbf{K}_{\boldsymbol{\theta}} + \sigma^2_{\text{noise}}\mathbf{I}\right|}_{\text{complexity penalty}} + \frac{n}{2}\log 2\pi. \tag{2.20}$$

The model fit term captures how well the output is described. Notice however that the complexity penalty only depends on the input and prior and thus acts as a regularizer! This built-in regularizing effect will be useful when combining it with neural networks in section 2.4.1.

There is of course the danger of over-fitting now that we are maximizing using the training data. However, this is usually not a problem with the kernels we have seen so far since the hyperparameters we optimize have broad implication for the function.

The maximization problem in itself is not trivial since it is non-convex. However, gradients of eq. (2.20) can be derived opening up for gradient based optimization approaches. Commonly the Quasi-Newton methods such as L-BFGS are used. Since we cannot hope to find a global maximum we instead seek a reasonable local maximum

which we attempt to find by starting the gradient based search at multiple random initial locationNs.

Whereas computing the posterior was only dominated by inverting $\mathbf{K}_{\boldsymbol{\theta}} + \sigma^2_{\text{noise}}\mathbf{I}$, the computation of the log marginal likelihood is restricted by both the inversion and also calculating the determinant of the same $N \times N$ matrix $\mathbf{K}_{\boldsymbol{\theta}} + \sigma^2_{\text{noise}}\mathbf{I}$. Section 2.2.2.5 and section 2.2.2.6 will provide two different solution to this.

We will sometimes refer to this optimization of hyperparameters as the *learning* step preceding *inference* in which we compute the posterior.

2.2.2.4 Automatic Relevance Detection

So far we have considered the SE kernel where the same length-scale is shared among all dimensions. For more flexibility we could consider optimizing a unique length-scale λ_d for each dimension

$$k_{\text{SE-ARD}}(\mathbf{x}, \mathbf{x}') = a^2 \exp\left[-\frac{1}{2}\sum_{d=1}^{D}\left(\frac{x_d - x'_d}{\lambda_d}\right)^2\right]. \tag{2.21}$$

This naturally leads to a way of selecting the relevant dimensions. To see why, notice that the optimal length-scale for an input dimension X_1, in which there is no change, is infinity. In other words, the term associated with X_1 will not contribute to the sum. For the prior this means that fixing all other dimensions, the function value becomes constant in the direction of X_1.

So this gives us a simple dimensionality reduction technique. The relevant dimensions have to be axis-aligned however. We will look at a more sophisticated attempt in section 3.3.1, but ARD will provide a good baseline.

2.2.2.5 Cholesky approach

For exact GPs, as well as the approximation we will see later in the chapter, the expressions include an inversion of a positive semi-definite matrix. However, usually we only have to solve the linear system $\mathbf{K}^{-1}\boldsymbol{y}$ such as when computing the posterior. The standard way of computing this is through the Cholesky decomposition which applies to a positive semi-definite matrix

$$\mathbf{K} = \mathbf{L}\mathbf{L}^\top \tag{2.22}$$

where $\mathbf{L}$ is a triangular matrix. It has the same running time as matrix inversion $\mathcal{O}(N^3)$ but is used because of its better constant factors and numerical stability.

2.2.2.6 Conjugate Gradient approach

Similar to Cholesky, the Conjugate Gradient (CG) method approaches the inversion by considering the matrix vector product in which $\boldsymbol{v}^* = (\mathbf{K} - \sigma^2\mathbf{I})^{-1}\boldsymbol{y}$ shows up [Shen et al.]. The crucial insight is that since the linear system of equation is positive definite, the solution to this system is also the unique minimum of the corresponding quadratic function $\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \left(1/2\boldsymbol{v}^\top\mathbf{K}\boldsymbol{v} - \boldsymbol{v}^\top\boldsymbol{u}\right)$. So instead we search for $\boldsymbol{v}^*$ by minimizing the quadratic function. CG provides an iterative scheme for this problem which converges to $\boldsymbol{v}^*$ within machine precision in t iterations. The exact number of iterations depends on how ill-conditioned the matrix $(\mathbf{K}-\sigma^2\mathbf{I})$ is. Using a *preconditioner* P we can improve the convergence by choosing it cleverly such that $P^{-1}(\mathbf{K} - \sigma^2\mathbf{I})$ is well-conditioned [Cutajar et al.]. Details lie outside the scope of this thesis but we will see in Appendix B.6 that choosing a sufficient size of this preconditioner is crucial.

This method leads to an $\mathcal{O}(tN^2)$ solution. An additional benefit of this approach is the fact that CG only requires a matrix-vector multiplication with $(\mathbf{K} - \sigma^2\mathbf{I})$ at every step. Because of this, the approach can be performed efficiently on a GPU.

2.3 Scalability

Some problems are intrinsically statistically challenging. That is, we need many observations to learn the underlying function. There are several reasons why this could be the case and one that we have already treated is dimensionality reduction, in section 3.3.1. Specifically framed in the context of GPs it also includes problems with high noise or low length-scale (relative to the domain)[1]. For high noise we would not be able to reduce the variance otherwise and as for the length-scale we need points sufficiently close to capture the variability. The computational bottleneck for GPs are thus a restriction on what problems we can tackle.

To scale GPs to large number of observations we have to overcome the computational bottleneck associated with the inverted kernel $\mathbf{K}_{NN}^{-1}$ popping up in inference and the determinant of $\mathbf{K}$ required for learning.

[1] [Bengio et al.] makes the effect of this function variability on the learning difficult precise.

2.3.1 Low Rank Approximation

As mentioned in section 2.2 inference for Gaussian processes has a computational complexity of $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2)$ which becomes prohibitively large for big data sets. This is because computing the posterior requires inversion of $\mathbf{K}_{NN} + \sigma^2_{\text{noise}}\mathbf{I}$ (or at least solving for $\boldsymbol{v}$ in $(\mathbf{K}_{NN} + \sigma^2_{\text{noise}}\mathbf{I})\boldsymbol{v} = \boldsymbol{y}$). So we are interested in finding a *low rank* approximation of $\mathbf{K}_{NN}$, assuming it indeed has full rank N, such that we only have to invert a lower rank matrix of rank $M < N$. If we leave aside the additive noise for the moment and only consider $\mathbf{K}_{NN}^{-1}$ such a low rank approximation can be achieved by truncating the eigendecomposition,

$$\mathbf{K}_{NN} \approx \tilde{\mathbf{K}}_{NN} = \mathbf{U}_{NM}\mathbf{\Lambda}_{MM}\mathbf{U}_{NM}^{\top}. \tag{2.23}$$

where $\mathbf{\Lambda}_M$ contains the first M eigenvalues and $\mathbf{U}_M$ the corresponding eigenvectors. In fact, this approximation is optimal with respect to the Frobinious norm and Spectral norm, i.e. it minimizes $||\mathbf{K}_{NN} - \tilde{\mathbf{K}}_{NN}||$ where $||\cdot||$ denotes the norm. To understand why this is desirable take the operator norm viewpoint (see section A.1.5) of the spectral norm and notice that it thereby bounds the worst case deviation. In other words, after applying $\tilde{\mathbf{K}}_{NN}$ we will always be close to $\mathbf{K}_{NN}\boldsymbol{x}$.

Now, this decomposition will allow fast inversion even when noise is added due to Woodburys inversion lemma (see section A.1.1) which for this particular case yields

$$(\mathbf{K}_{NN} + \sigma_\epsilon^2\mathbf{I}_N)^{-1} = \sigma_\epsilon^{-2}\mathbf{I}_N + \sigma_\epsilon^{-2}\mathbf{U}_{NM}\left(\sigma_\epsilon^2\mathbf{\Lambda}_{mm}^{-1} + \mathbf{U}_{NM}^{\top}\mathbf{U}_{NM}\right)^{-1}\mathbf{U}_{NM}^{\top}. \tag{2.24}$$

Notice, crucially, that we have now reduced it to inversion of a smaller matrix of size $M \times M$. Further the determinant needed for the marginal log likelihood calculation used for hyperparameter optimization in eq. (2.20) can be calculated with the Sylvester determinant theorem (section A.1.2)

$$|\tilde{\mathbf{K}}_{NN} + \sigma_{\text{N}}^2\mathbf{I}| = \left|\mathbf{\Lambda}_{mm}\|\sigma_\epsilon^2\mathbf{\Lambda}_{MM}^{-1} + \mathbf{U}_{NM}^{\top}\mathbf{U}_{NM}\right|. \tag{2.25}$$

Computations have now similarly been reduced for the determinant by only requiring $\mathcal{O}(M)$ for inverting the diagonal matrix of size $M \times M$ and $\mathcal{O}(M^3)$ for calculating the determinant in the expression.

So, for a given M, the optimal approximate inference can be done in $\mathcal{O}(NM^2 + M^3)$ since the computational bottleneck is the matrix-matrix multiplication of an $M \times M$ and $M \times N$ matrix and the matrix inversion. The catch is that eigendecomposition in

itself is a $\mathcal{O}(N^3)$ operation so we do not gain anything computationally. However, it does suggest the existence of a good approximation and its possible form.

2.3.1.1 Inducing points approximation

One popular family of approximation methods that achieves scalability through this low rank approximation is the *inducing point* methods. We will take a slightly difference view than that of the kernel to develop it. To simplify notation slightly we will implicitly assume conditioning on the input variables.

The general idea is to rewrite the joint prior $p(\boldsymbol{f}_T, \boldsymbol{f})$ by including an additional set of M latent variables $\boldsymbol{u} = [u_1, \ldots, u_M]^\top$ such that we consider $p(\boldsymbol{f}_T, \boldsymbol{f}, \boldsymbol{u})$ instead [Quiñonero-Candela and Rasmussen]. Notice that we can recover our original joint prior by simply marginalizing out $\boldsymbol{u}$

$$p(\boldsymbol{f}_T, \boldsymbol{f}) = \int p(\boldsymbol{f}_T, \boldsymbol{f}, \boldsymbol{u}) \, \mathrm{d}\boldsymbol{u} = \int p(\boldsymbol{f}_T, \boldsymbol{f}|\boldsymbol{u}) \, p(\boldsymbol{u}) \mathrm{d}\boldsymbol{u} \quad \text{where} \quad p(\mathbf{u}) = \mathcal{N}(\mathbf{0}, \mathbf{K}_{MM}) .$$

So far the prior is still exact. To turn it into an approximation (and by that hopefully achieve computational benefits) we need to put down some assumption. Almost all inducing point methods assume conditional independence between $\boldsymbol{f}_T$ and $\boldsymbol{f}$ given $\boldsymbol{u}$ so that the integrand factorises [Quiñonero-Candela and Rasmussen],

$$p(\boldsymbol{f}_T, \boldsymbol{f}) \approx q(\boldsymbol{f}_T, \boldsymbol{f}) = \int q(\boldsymbol{f}_T|\boldsymbol{u}) \, q(\boldsymbol{f}|\boldsymbol{u}) p(\boldsymbol{u}) \mathrm{d}\boldsymbol{u}.$$

This has a nice intuitive interpretation in that $\boldsymbol{f}$ and $\boldsymbol{f}_T$ are only dependent on each other through $\boldsymbol{u}$. In that sense $\boldsymbol{u}$ acts as a *bottleneck* for the dependency.

The different inducing point methods differ by further placing various simplifying assumptions on $p(\boldsymbol{f}|\boldsymbol{u})$ and $p(\boldsymbol{f}_T|\boldsymbol{u})$ to get approximations denoted by $q(\boldsymbol{f}|\boldsymbol{u})$ and $q(\boldsymbol{f}_T|\boldsymbol{u})$ respectively. Interpreting $\boldsymbol{u}$ as observation and $\boldsymbol{f}$ as well as $\boldsymbol{f}_T$ as predictions we easily write down the mean function and kernel of the exact form by reusing the posterior developed in section 2.2,

$$\begin{aligned} \text{Training conditional:} \quad & p(\boldsymbol{f}|\boldsymbol{u}) = \mathcal{N}\left(\mathbf{K}_{NM}\mathbf{K}_{MM}^{-1}\boldsymbol{u}, \mathbf{K}_{NN} - \mathbf{Q}_{NN}\right), \\ \text{Test conditional:} \quad & p(\boldsymbol{f}_T|\boldsymbol{u}) = \mathcal{N}\left(\mathbf{K}_{TM}\mathbf{K}_{MM}^{-1}\boldsymbol{u}, \mathbf{K}_{TT} - \mathbf{Q}_{TT}\right), \end{aligned} \tag{2.26}$$

where $\mathbf{Q}_{AB} \triangleq \mathbf{K}_{AM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MB}$.

2.3.2 Subset of Regressors

One of the simplest assumptions we can place on the test and training conditionals in eq. (2.26) is $\mathbf{K}_{NM} - \mathbf{Q}_{NN} = \mathbf{0}$ and $\mathbf{K}_{MT} - \mathbf{Q}_{TT} = \mathbf{0}$ such that

$$\begin{aligned} q(\boldsymbol{f}|\boldsymbol{u}) &= \mathcal{N}\left(\mathbf{K}_{NM}\mathbf{K}_{MM}^{-1}\boldsymbol{u}, \mathbf{0}\right), \\ q\left(\boldsymbol{f}_T|\boldsymbol{u}\right) &= \mathcal{N}\left(\mathbf{K}_{TM}\mathbf{K}_{MM}^{-1}\boldsymbol{u}, \mathbf{0}\right). \end{aligned} \tag{2.27}$$

To start making predictions we first need to write down the joint distribution $p\left(\mathbf{f}, \mathbf{f}_T\right)$ as done for the noise-free setting in section 2.2.2.1. Observe that because of the independence assumption $\boldsymbol{f} \perp\!\!\!\perp \boldsymbol{f}_T$ we can write

$$q(\boldsymbol{f}|\boldsymbol{u})p(\boldsymbol{f}_T|\boldsymbol{u}) = q(\boldsymbol{f}, \boldsymbol{f}_T|\boldsymbol{u}) = N(\mathbf{K}_{(T+N)M}\mathbf{K}_{MM}\boldsymbol{u}, \mathbf{0}). \tag{2.28}$$

which allows us to readily marginalize out $\boldsymbol{u}$ (see appendix section A.1.4)

$$p\left(\mathbf{f}, \mathbf{f}_T\right) \approx q_{\mathrm{SoR}}\left(\mathbf{f}, \mathbf{f}_T\right) = \int q(\boldsymbol{f}|\boldsymbol{u}) q\left(\boldsymbol{f}_T|\boldsymbol{u}\right) p(\boldsymbol{u}) d\boldsymbol{u} = \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{Q}_{NN} & \mathbf{Q}_{NT} \\ \mathbf{Q}_{TN} & \mathbf{Q}_{TT} \end{bmatrix}\right), \tag{2.29}$$

where we recall that $\mathbf{Q}_{AB} \triangleq \mathbf{K}_{AM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MB}$.

We can now proceed as usual to get the approximate posterior (using section A.1.4),

$$q(\boldsymbol{f}_T|\boldsymbol{f}) = \mathcal{N}\left(\mathbf{Q}_{TN}\mathbf{Q}_{NN}^{-1}\boldsymbol{f}, \mathbf{Q}_{TT} - \mathbf{Q}_{TN}\mathbf{Q}_{NN}^{-1}\mathbf{Q}_{NT}\right). \tag{2.30}$$

The only difference from the posterior in eq. (2.11) for the noise-free setting, is that all the blocks of the kernels have now been approximated $\mathbf{K}_{AB} \approx \mathbf{Q}_{AB} = \mathbf{K}_{AM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MB}$. One limitation of this is apparent if we consider a stationary kernel with $k(\boldsymbol{x} - \boldsymbol{x}')$ decreasing as the distance $|\boldsymbol{x} - \boldsymbol{x}'|$ increases. Then $\mathbf{Q}_{TT}$ will be close to zero if the M inducing points are far from the T test points since entries in $\mathbf{K}_{MT}$ will be close to zero. So the variance of the test set will be underestimated away from the inducing points. We will notice a similar problem for the more abstract sparse spectrum inducing point scheme in section 2.3.5.

The whole motivation for introducing these approximating assumptions was to reduce the complexity of inverting the kernel $\mathcal{O}(N^3)$. The kernel $\mathbf{K}_{NN}^{-1}$ has now been replaced by $\mathbf{Q}_{NN}^{-1} = \left(\mathbf{K}_{NM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MN}\right)^{-1}$ which in the noisy setting using Woodbury's inversion lemma (see section A.1.1) can be reduced to only inverting the smaller $\mathbf{K}_{MM}^{-1}$. We thus obtain a low rank approximation taking $\mathcal{O}(NM^2 + M^3)$ as argued in section 2.3.1.

2.3.3 Variational Sparse Gaussian Processes

So far we have not talked about selecting the inducing points. An elegant method that does so is that of the Variational Free-Energy [Titsias]. It places no further assumptions than the original inducing point assumption: that $\boldsymbol{u}$ is still a bottleneck in the sense that $\boldsymbol{f}_T$ is independent of $\boldsymbol{f}$ given $\boldsymbol{u}$. Instead it directly approximates the posterior by finding a distribution $q(\boldsymbol{u})$ on $\boldsymbol{u}$ chosen to be *any* gaussian distribution. We will give a high-level introduction of the method here.

To find suitable hyperparameters to compute the posterior we saw in section 2.2.2.3 that we could maximize the marginal log likelihood

$$\ln p(\boldsymbol{y}|\boldsymbol{\theta}) = \ln \int p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta}) \mathrm{d}\boldsymbol{f}. \tag{2.31}$$

The key idea is now to introduce some distribution $q(\boldsymbol{u})$ and observe that we can write the integral as an expectation,

$$\ln p(\boldsymbol{y}|\boldsymbol{\theta}) = \ln \int p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta}) \frac{q(\boldsymbol{f})}{q(\boldsymbol{f})} \mathrm{d}\boldsymbol{f} = \ln \mathbb{E}_{q(\boldsymbol{f})} \left[\frac{p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta})}{q(\boldsymbol{f})} \right], \tag{2.32}$$

We can bound this quantity by Jensen's inequality which allows us to push the logarithm into the expectation such that

$$\ln p(\boldsymbol{y}|\boldsymbol{\theta}) \geq \mathbb{E}_{q(\boldsymbol{f})} \left[\ln \frac{p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta})}{q(\boldsymbol{f})} \right]. \tag{2.33}$$

By applying the product rule we can split the expectation into two terms

$$\ln p(\boldsymbol{y}|\boldsymbol{\theta}) \geq \mathbb{E}_{q(\boldsymbol{f})} \left[\ln \frac{p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta})}{q(\boldsymbol{f})} \right] = \ln p(\boldsymbol{y}|\boldsymbol{\theta}) - \mathbb{E}_{q(\boldsymbol{f})} \left[\ln \frac{q(\boldsymbol{f})}{p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta})} \right], \tag{2.34}$$

where the last term is known as the Kullback–Leibler (KL) divergence $\mathrm{KL}(q(\boldsymbol{f}) \parallel p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta})) = \mathbb{E}_{q(\boldsymbol{f})} \left[\ln \frac{q(\boldsymbol{f})}{p(\boldsymbol{y}, \boldsymbol{f}|\boldsymbol{\theta})} \right]$. This is a well-studied quantity in Information Theory and is known to be positive. So to maximize the log marginal likelihood over the distribution $q(\boldsymbol{f})$ we want to minimize the KL-divergence. Intuitively the KL-divergence provides a measure of closeness between two distributions. So by minimizing the expression we are finding a $q(\boldsymbol{f})$ that most closely approximates $p(\boldsymbol{y}, f|\boldsymbol{\theta})$.

Up until now this is the general method of Variational Inference. Without restricting $q(\boldsymbol{f})$ we will simply obtain the true log maximum likelihood. We will now use our assumption to pick a particular family for $q(\boldsymbol{f})$ which makes it computation tractable. With slight abuse of notation, our independence assumption allows us to factorize

$q(\boldsymbol{f}, \boldsymbol{u}) = p(\boldsymbol{f}|\boldsymbol{u})q(\boldsymbol{u})$. We pick $q(\boldsymbol{u})$ to be "free" in the sense that it can be any Gaussian distribution. The optimization problem is then to optimize for this Gaussian distribution.

Several details have been left out such as how to actually maximize the KL-divergence for which we refer to [Titsias]. From here-on we will refer to the method as SGPR, for Sparse Gaussian Process Regression.

2.3.4 Kernel Interpolation for Scalable Structured Gaussian Processes

While the inducing point method achieves scalability in N using $\mathcal{O}\left(M^2N + M^3\right)$ computations and $\mathcal{O}\left(MN + M^2\right)$ storage it is constrained to choosing $M \ll N$ to be computationally tractable. This provides a restriction on what we can learn even as the dataset grows.

Alternatives to inducing point methods have been proposed that instead restrict the kernel to a particular family such as Kronecker [Saatci] and Toeplitz [Cunningham et al.]. This structure can then be exploited for scalable inference. Intuitively, assuming this kernel prior is well picked, we will be able to leverage the numerous observations at our disposal much better. This is particularly interesting in the financial setting where we expect local features which can only be discovered by observation.

However these methods require the observations to live on a cartesian product grid. This goes against one of our original motivations for moving away from the state-of-the-art Adaptive Sparse Grid approaches to the value iteration problem in the first place. To allow for arbitrary observations one could naively combine this approach with inducing points by instead inducing the grid points. However, this will only speed-up the matrix inversion thus taking care of the term $\mathcal{O}(M^3)$ in the computational complexity.

To deal with the term $\mathcal{O}\left(NM^2\right)$ associated with the matrix-matrix multiplication [Wilson and Nickisch] approximate $\mathbf{K}_{NM}$ by interpolating $\mathbf{K}_{MM}$. Combined with Kronecker and Toeplitz structure by picking inducing points on a grid this leads to a scalable approach with linear running time in N: $\mathcal{O}(N + M \log M)$ for 1D and $\mathcal{O}\left(N + DM^{1+1/D}\right)$ for higher dimensions. This method is referred to as Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP).

2.3.4.1 Toeplitz structure

If our kernel is a stationary one, $k(x, x') = k(x - x')$, defined on scalar input and these inputs $\{ x_i \}_{i=1}^N$ are spaced on a regular grid then the associated covariance matrix $\mathbf{K}_{NN}$ is a Toeplitz matrix. That is, $\mathbf{K}_{NN}$ is a matrix with constant diagonals $\mathbf{K}_{NN}(i, j) = \mathbf{K}_{NN}(i + 1, j + 1)$. This is easy to see, as the ith row of $\mathbf{K}_{NN}$ contains all $k(x_i, x_j)$ for $j = 1, ..., N$. So each row only differs by being centered around a different x_i. Toeplitz matrices are well-studied in signal processing and application to GPs is discussed in [Fritz et al.].

It turns out that we can embed this matrix in a circulant matrix on which Fast Fourier Transformation can be used to compute the vector-matrix product. This computation is sufficient to compute $(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\boldsymbol{y}$ using the CG iterations approach in section 2.2.2.6.[2] This allows us to do the otherwise expensive kernel inversions in $\mathcal{O}(M \log M)$.

2.3.4.2 Kronecker structure

If we are in higher dimensions than 1D we need to exploit some other structure than Toeplitz structure. Suppose our kernel decomposes into a product of kernels defined on each input dimension,

$$k(\mathbf{x}_i, \mathbf{x}_j) = \prod_{d=1}^{D} k\left(\mathbf{x}_i^{(d)}, \mathbf{x}_j^{(d)}\right), \tag{2.35}$$

which is the case for, e.g., the SE kernel. If we now cleverly place our inputs on a cartesian grid[3] $\{ \boldsymbol{x} \}_{i=1}^N = \{ \mathbf{x}_i^{(1)} \}_{j=1}^n \times \cdots \times \{ \mathbf{x}_i^{(D)} \}_{j=1}^n$ then the corresponding covariance matrix $\mathbf{K}$ can be split into a Kronecker product of matrices over each input dimension

$$\mathbf{K} = \mathbf{K}_1 \otimes \cdots \otimes \mathbf{K}_D, \tag{2.36}$$

where the Kronecker product is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}. \tag{2.37}$$

[2] [Chan and Ng] discusses Toeplitz matrices in the context of GPs thoroughly.

[3] We note in passing that there is nothing preventing us from choosing a different granularity of the grid than n for each dimensions but we will not need this flexibility.

This means that the kernel has now been decomposed into matrices of the much lower dimensionality n.

We saw in section 2.3.1 that if we can do eigendecomposition then inversion becomes fast. Fortunately, since each $\mathbf{K}_i$ is positive semi-definite they all admit an eigendecomposition. More crucially, because of the mixing property[4] in Kronecker algebra we can rewrite the eigendecomposition of $\mathbf{K}$ in terms of the eigendecomposition of these smaller matrices

$$\mathbf{K} = \mathbf{K}_1 \otimes \cdots \otimes \mathbf{K}_D = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top = (\mathbf{Q}_1 \otimes \cdots \otimes \mathbf{Q}_D)(\mathbf{\Lambda}_1 \otimes \cdots \otimes \mathbf{\Lambda}_D)(\mathbf{Q}_1 \otimes \cdots \otimes \mathbf{Q}_D)^\top, \quad (2.38)$$

where the eigendecomposition of the smaller matrices is as expected $\mathbf{K}_d = \mathbf{Q}_d\mathbf{\Lambda}_d\mathbf{Q}_d^\top$.

We have seen in section 2.3.1 that we can exploit the eigendecomposition to write $(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\boldsymbol{y} = \mathbf{Q}\left(\mathbf{V} + \sigma^2\mathbf{I}\right)^{-1}\mathbf{Q}^T\boldsymbol{y}$ in the noisy setting. However, while $\mathbf{\Lambda}$ is trivial to compute and invert, $\mathbf{Q}$ is unknown. Fortunately, it turns out that we can perform fast matrix vector multiplication if the matrix has Kronecker structure so $\mathbf{Q}^\top\boldsymbol{y}$ can be computed through its Kronecker decomposition. The log determinant used for hyperparameter learning can similarly be sped up. Overall, this leads to a time complexity of $\mathcal{O}\left(DM^{1+\frac{1}{D}}\right)$ and space complexity of $\mathcal{O}\left(DM^{\frac{2}{D}}\right)$ for both inference and learning. For a detailed treatment we refer to [Saatci].

2.3.4.3 Structured Kernel Interpolation

We will now address the computational complexity $\mathcal{O}(NM^2)$ of the matrix-matrix multiplications in our approximation $\mathbf{K}_{NN} \approx \mathbf{K}_{NM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MN}$ (developed in eq. (2.29)) as proposed in [Wilson and Nickisch]. The computational speed-up is achieved through approximating $\mathbf{K}_{NM}$ by locally interpolating on the $\mathbf{K}_{UU}$ matrix.

To see how the interpolation is done, let us first assume $\mathbf{K}_{MM}$ is Toeplitz with points living in 1D spaced on a regular grid. To approximate $k(\boldsymbol{x}_i, \boldsymbol{u}_j)$ we find $\boldsymbol{u}_a$ and $\boldsymbol{u}_b$ that most closely bounds $\boldsymbol{x}_i$. Then we interpolate linearly between them $\tilde{k}\left(\boldsymbol{x}_i, \boldsymbol{u}_j\right) = w_i k\left(\boldsymbol{u}_a, \boldsymbol{u}_j\right) + (1 - w_i)\, k\left(\boldsymbol{u}_b, \boldsymbol{u}_j\right)$ where the weight w_i is the relative distance from $\boldsymbol{x}_i$ to $\boldsymbol{u}_a$ and $\boldsymbol{u}_b$. In matrix form this would be

$$\mathbf{K}_{NM} \approx \mathbf{W}\mathbf{K}_{MM}, \quad (2.39)$$

where each row i in $\mathbf{W}$ only has two non-zero entries with one being w_i and the other being $(1 - w_i)$. So $\mathbf{W}$ becomes extremely sparse because of this local interpolation and

[4]The mixing property states that $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$. For more detail we refer to [Saatci] specifically on Kronecker structure for GPs.

consequently allows for fast matrix-matrix multiplication. Following the suggestion in the original work [Wilson and Nickisch] we use cubic interpolation instead which still only requires 4 non-zero entries per row.

Using this approximation in our inducing point method we obtain

$$\begin{aligned}\mathbf{K}_{NN} &\overset{\text{SoR}}{\approx} \mathbf{K}_{NM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MN} \\ &\overset{\text{SKI}}{=} \mathbf{W}\mathbf{K}_{MM}\mathbf{K}_{MM}^{-1}\mathbf{K}_{MM}\mathbf{W}^T = \mathbf{W}\mathbf{K}_{MM}\mathbf{W}^\top = \mathbf{K}_{\text{SKI}},\end{aligned} \tag{2.40}$$

abbreviated SKI for *structured kernel interpolation*. We can now exploit the sparsity of $\mathbf{W}$ in $\mathbf{K}_{\text{SKI}}$ to compute the vector-matrix multiplication required in the posterior GP in only $\mathcal{O}(N + M^2)$.

For grid structured inducing point we can naturally extend this to higher dimensions by some weighting scheme on the set of inducing point that makes out the tightest hypercube surrounding $\boldsymbol{x}_i$. In that case $\mathbf{W}$ will have 2^D non-zero elements for each row for linear interpolation.

2.3.5 Sparse Spectrum Gaussian Processes

As mentioned in section 2.3.1.1 more imaginative choices of $\boldsymbol{u}$ exist than ones coming from the input domain. One such choice which we will now develop is using the frequency domain which will be connected to the inducing points framework eventually.

We will take a stationary kernel and approximate it with finite many samples from the frequency spectrum. The key insight that will allow this is Bochner's Theorem which connects positive definite kernels to their Fourier transformation.

Theorem 2.3.1 (Bochner [Rudin])**.** *A continuous stationary kernel $k(\boldsymbol{x}, \boldsymbol{y}) = k(\boldsymbol{x} - \boldsymbol{y})$ on $\mathcal{R}^D$ is positive definite iff k is the Fourier transform of a normalized probability distribution*

$$k(\boldsymbol{x} - \boldsymbol{y}) = \mathcal{F}\{p(\boldsymbol{\omega})\}. \tag{2.41}$$

We know that if k is properly-scaled (that is $k(\boldsymbol{x}, \boldsymbol{x}) = 1$ for all $\boldsymbol{x}$) then its Fourier transformation is a normalized probability distribution. By Bochner's theorem we can then write k as an integral over its Fourier transform $s(\boldsymbol{\omega})$

$$k(\boldsymbol{x} - \boldsymbol{y}) = k(\boldsymbol{\tau}) = \mathcal{F}^{-1}\{s(\boldsymbol{\omega})\} = \int_{\mathbb{R}^D} s(\boldsymbol{\omega}) \exp\left(i\boldsymbol{\omega}^\top \boldsymbol{\tau}\right) \mathrm{d}\boldsymbol{\omega}, \tag{2.42}$$

where

$$s(\boldsymbol{\omega}) = \mathcal{F}\{k(\boldsymbol{\tau})\} = \int_{\mathbb{R}^D} k(\boldsymbol{\tau}) \exp\left(-i\boldsymbol{\omega}\boldsymbol{\tau}\right) \mathrm{d}\boldsymbol{\tau} \text{ and } \boldsymbol{\tau} = \boldsymbol{x} - \boldsymbol{y}. \tag{2.43}$$

We can interpret this as an expectation over the $\boldsymbol{\omega}$,

$$k(\boldsymbol{x}-\boldsymbol{y}) = \mathbb{E}_{\omega}\left[\exp\left(i\boldsymbol{\omega}^{\top}(\boldsymbol{x}-\boldsymbol{y})\right)\right] = \mathbb{E}_{\omega}\left[\left\langle \exp\left(i\boldsymbol{\omega}^{\top}\boldsymbol{x}\right), \exp\left(i\boldsymbol{\omega}^{\top}\boldsymbol{y}\right)\right\rangle\right]. \tag{2.44}$$

The idea is to approximate this expectation with some weighted sum over $\bar{M}$ frequencies which we can then write as a Hermitian inner product

$$k(\boldsymbol{x}-\boldsymbol{y}) \approx \sum_{j=1}^{\bar{M}} a_j \left\langle \exp\left(i\boldsymbol{\omega}_j^{\top}\boldsymbol{x}\right), \exp\left(i\boldsymbol{\omega}_j^{\top}\boldsymbol{y}\right)\right\rangle = \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{y}), \tag{2.45}$$

where a_j is the non-negative weight associated with each $\boldsymbol{\omega}_j$ and $\boldsymbol{\phi}(\boldsymbol{x})_j = \sqrt{a_j}\exp\left(\pi i\boldsymbol{\omega}_j^{\top}\boldsymbol{x}\right)$.

Notice how the kernel approximation is now a linear kernel with only $\bar{M}$ entries. Expressed on matrix form the covariance matrix decomposes into $\mathbf{K} = \boldsymbol{\Phi}^{\top}\boldsymbol{\Phi}$ where $\boldsymbol{\Phi} = [\boldsymbol{\phi}(\boldsymbol{x}_1)\cdots\boldsymbol{\phi}(\boldsymbol{x}_N)]$. Since $\boldsymbol{\Phi}$ is $\bar{M}\times N$ this yields a low rank approximation so it allows for fast inversion if $\bar{M} \ll N$.

How to pick $\{\,(a_j,\boldsymbol{\omega}_j)\,\}_{j=1}^{M}$. What is left is deciding how to pick the M spectral points and the associated weights $\{\,a_j\,\}_{j=1}^{M}$. One celebrated approach uses Monte Carlo sampling of $p(\omega)$ effectively picking random Fourier features [Rahimi and Recht]. This is a simple way of achieving an unbiased estimator.

In addition we will also consider optimizing these locations in the spectral domain as proposed in [McIntire et al.]. This is achieved by simply absorbing them into the hyperparameters. One should be careful though, as this might lead to overfitting because we are now optimizing instead of integrating. A further complication is that the already non-convex optimization over the marginalized log-likelihood now becomes high-dimensional as well.

More sophisticated techniques to approximating the integral have been considered but we will leave the review of them for the discussion in **??**. This leaves us with two approaches: the Monte Carlo based variant which we will denote SSGP-FIXED and the optimized variant SSGP – both being an abbreviation of Sparse Spectrum Gaussian Processes.

The SE kernel So far we have described the approach in abstract mathematical terms and not given a solution to how to evaluate $\boldsymbol{\phi}(\boldsymbol{x})$ containing imaginary terms. To fill this gap we will consider our favorite stationary kernel, the SE, which is indeed also the kernel we will use in our experiments. For later reference the SE kernel has

the following form in the time and frequency domains respectively:

$$\begin{aligned} K_{\text{SE}}(\boldsymbol{\tau}) &= \exp\left(-\boldsymbol{\tau}^2/\ell^2\right) \quad \text{and} \\ S_{\text{SE}}(\boldsymbol{\omega}) &= 2\pi\ell^2 \exp\left(-2\pi^2\ell^2\boldsymbol{\omega}^2\right). \end{aligned} \tag{2.46}$$

Because the kernel is symmetric it simplifies to a sum of cosines for which we can use a trigonometric identity to split it into a dot product (see **??** for details). We obtain

$$\boldsymbol{\phi}(\boldsymbol{x})_{i=1}^{\bar{M}} = (\cos 2\pi\boldsymbol{x}\omega_i, \sin 2\pi\boldsymbol{x}\omega_i). \tag{2.47}$$

Note that this implies that if we choose to approximate a GP using $\bar{M}$ spectral points this would effectively lead to computations with vectors of size $M = 2\bar{M}$. Since we are interested in comparing model performance for a fixed computational budget we will from now on specify the SSGP method in terms of the *actual size* M.

Implementation details In practice we will be optimizing over the length-scale meaning that the distribution in section 2.3.5 from which we sample $\boldsymbol{\omega}$ will change. However, this would complicate matters since the function we optimize would change for every update of the length-scale leaving us with a stochastic optimization problem. Instead we want the samples fixed which we achieve by observing that we can simply rescale the distribution based on the length-scale. For the SE kernel in particular we can sample $\boldsymbol{\omega}'$ from a standard normal distribution $\boldsymbol{\omega}'_i \sim \mathcal{N}(0, 1)$ and rescale such that $\boldsymbol{\omega} = \frac{1}{\ell}\boldsymbol{\omega}'$. This extends without complications to higher dimensions with an ARD kernel where we have a unique length-scale for each dimensions $\boldsymbol{\omega} = \boldsymbol{\Lambda}\boldsymbol{\omega}'$.

2.4 High-dimensionality

Regression problems when the dimensionality D is large are intrinsically hard statistical problems. That is because the number of samples we need to learn a D multivariate distribution grows exponentially with D. To see this consider the points in a grid needed to densely cover a high-dimensional space. This cartesian product grid grows exponentially in the dimensionality. Specifically euclidean distance, which GP regression with stationary kernels rely on, becomes uninformative as the input dimensionality grows. We can only hope to address this curse of dimensionality if the objective function has some special structure that we can discover and exploit.

We thus need some assumption to reduce the *effective dimensionality* of the problem. One common assumption is that the input lives on a low dimensional manifold in the

higher dimensional space. To be precise we assume that the objective function f can be decomposed into

$$f = g \circ h \tag{2.48}$$

where the deterministic mapping $h : \mathcal{X} \to \mathcal{L}$ takes the input into a lower dimensional *feature space* $\mathcal{L}$ from where there is a probablistic relationship with the output space $g : \mathcal{L} \to \mathcal{Y}$.

In a fully Bayesian framework we would treat $\mathcal{L}$ as a latent space and integrate it out. However, this would be intractable in most cases. One approximation would be to learn h in a greedy fashion before g, in which case it is necessary to define some unsupervised objective. Even though successful they suffer in cases where the unsupervised objective does not match with the supervised objective of maximizing the marginal likelihood [Calandra et al.].

In the financial setting we see this misalignment as it is desirable to learn a feature space that is smooth such that GP regression works well on this transformed space. To this end we consider jointly learning the low-dimensional embedding of original data and the probabilistic mapping from low-dimensional space to observation space as proposed in [Calandra et al.] and [45] by the name of Manifold Gaussian Processes and Deep Kernel Learning respectively.

Remark A much more crucial drawback about unsupervised approaches is that there is no structure to be found if the data is uniformly distributed in the high dimensional space. In other words, supervised methods are strictly necessary if the data is uniform. Much of the literature fail to make this distinction between when a mapping into a lower dimensional space is assumed to exists (such as in our case) and when the data lives on a low dimensional manifold *embedded* into the high dimensional space. This is especially important to point out since the name *Manifold* Gaussian Processes might be misleading.

We need support on the whole high dimensional space and thus cannot assume that data only lies on an embedded manifold. Take for example the Dynamic Economic Modelling problem introduced in section 2.1 in which each dimension represents an agent. We cannot simply leave out certain configurations of the agents.

Since our data is uninformative about the structure we need the function $f(\boldsymbol{x})$ to guide the learning of the feature mapping h in a supervised manor. On such method, that we will consider, called *Active Subspaces* learns a linear mapping to a lower dimensional space. It is guided by f through its gradient and does not learn a mapping to the output space. It is important to note that even though it is supervised it is

still not aligned with our final objective of estimating $f(\boldsymbol{x})$. Thus it might suffer compared with Manifold Gaussian Processes even if we disregarded the stronger (linear) assumption it makes on h.

If we were to put restrictions on the distribution data in the input space a whole range of unsupervised techniques could be considered. One of the simplest approaches is a linear projection chosen randomly. More clever choices of the linear projection includes maximization of the variance (PCA) and maximization of the statistical independence (ICA). If we were to relax our assumption further and consider embedded non-linear manifolds techniques exists that preserves distance (Isomap) or minimizes the input reconstruction error (Autoencoders). Autoencoders further leads to interesting semi-supervised methods in combination with GPs that we discuss in chapter 5.

We leave the overview here since they might become relevant in settings where the input data actually lives on a manifold.

2.4.1 Deep Kernel Learning

Instead of applying GP regression directly to learn $f : \mathcal{X} \to \mathcal{Y}$ we learn a GP on the feature space $g : \mathcal{L} \to \mathcal{Y}$ using the transformed inputs $\mathbf{H} = h(\mathbf{X})$. This is equivalent to a GP for f with a covariance function

$$k(\boldsymbol{x}, \boldsymbol{x}') = \tilde{k}(h(\boldsymbol{x}), h(\boldsymbol{x}')), \tag{2.49}$$

where $\tilde{k}$ has hyperparameters $\tilde{\boldsymbol{\theta}}$. If we approximate h with a learned mapping $\phi : \mathcal{X} \to L$ parameterized by $\boldsymbol{w}$ then we can consider these hyperparameters as part of the kernel and optimize $\boldsymbol{\theta} = \{\, \tilde{\boldsymbol{\theta}}, \boldsymbol{w} \,\}$ jointly through the marginal likelihood.

2.4.1.1 Marginal Log Likelihood

Recall that to maximize the log marginal likelihood from section 2.2.2.3 we make use of the gradient. Obtaining the analytical gradient is indeed still possible for deep kernels. We need to be able to compute the partial derivatives with respect to both $\tilde{\boldsymbol{\theta}}$ and $\boldsymbol{w}$. We can simply back-propagate through the marginal likelihood—that is, apply the chain rule.

$$\frac{\partial \operatorname{LML}(\boldsymbol{\theta})}{\partial \tilde{\boldsymbol{\theta}}} = \frac{\partial \operatorname{LML}(\boldsymbol{\theta})}{\partial \mathbf{K}_{\boldsymbol{\theta}}} \frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \tilde{\boldsymbol{\theta}}}, \quad \frac{\partial \operatorname{LML}(\boldsymbol{\theta})}{\partial \mathbf{w}} = \frac{\partial \operatorname{LML}(\boldsymbol{\theta})}{\partial \mathbf{K}_{\boldsymbol{\theta}}} \frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \mathbf{H}} \frac{\partial \mathbf{H}}{\partial \mathbf{w}} \tag{2.50}$$

where LML denotes the log marginal likelihood and $\mathbf{H} = h(\mathbf{X})$ is the deterministic feature mapping.

2.4.1.2 Choice of input transformation h

Any deterministic mapping could be used $\boldsymbol{\phi} : \mathcal{X} \to \mathcal{L}$ but we will focus on parameterization by multi-layer neural networks as these have shown to be expressive (i.e. universal approximators [Hornik et al.]). We will refer to a specific network architecture as $[q_1\text{-}\ldots\text{-}q_l]$ where l is the number of layers and q the number of hidden units in each layer. Specifically $[q_1\text{-}\cdots\text{-}q_n]$ will denote the transformation

$$M(\boldsymbol{X}) = (\mathbb{T}_l \circ \ldots \circ \mathbb{T}_1)(\boldsymbol{X}), \tag{2.51}$$

where each layer $i = 1, ..., l$ applies the transformation $\mathbb{T}_i(\boldsymbol{Z}) = \sigma_i(\boldsymbol{W}_i\boldsymbol{Z} + \boldsymbol{B}_i)$ with $\mathbf{W}_i$ and $\mathbf{B}_i$ such that the output dimensionality is q_i. The mapping $\sigma_i(\cdot)$ for $i = 1, ..., l-1$ will be some activation function specified by the context and σ_l is the identity function. So [100-50-2] denotes a fully connected network for 2 hidden layers of 100 hidden units followed by 50 hidden units with an output layer of 2 units. After each transformation an activation function is applied except for the last.

Our parameterization is then $\boldsymbol{w} = [\mathbf{W}_1, \mathbf{B}_1, \ldots, \mathbf{W}_l, \mathbf{B}_l]$. The gradient for each of these are calculated with back-propagation in the usual way through repeated application of the chain rule.

2.4.2 Scalable Deep Kernel Learning

Compare to a stand-alone neural network, each gradient step in DKL is significantly more expensive. This is because of the $\mathcal{O}(N^3)$ computation required for the log marginal likelihood and its derivative which severely restrict what it is capable of learning. We will now consider the approximate methods from section 2.3 on scalability and a particular choice of GP kernel to overcome this challenge.

2.4.2.1 Using Approximate Methods

The original paper on DKL proposes combining this with KISS-GP from section 2.3.4 to immediately achieve linear scaling in N, $\mathcal{O}\left(N + DM^{1+1/D}\right)$ time complexity. However, as the complexity term captures, this restricts the output dimensionality of the network: in practice we are limited to $D \leq 5$.

Alternatively, we will also consider the inducing point methods SGPR (section 2.3.3) and SSGP (section 2.3.5) which scales as $O(NM^2 + M^3)$. For intermediate sizes of observations ($10^4 < N < 10^5$) these tends to run faster than KISS-GP even though they are asymptotically slower.

2.4.2.2 Bayesian Linear Regression

Instead of approximating the kernel we could instead assume the kernel to be linear,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \tilde{k}_{\text{linear}}(\boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')) = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{x}'). \tag{2.52}$$

As briefly mentioned in section 2.2.1.1, a GP with a linear kernel turns out to be equivalent to a bayesian linear regressor [Rasmussen and Williams]. With this restriction on the kernel, both learning and inference can be performed in $\mathcal{O}(Nd^2)$ where d is the dimensionality of the feature space.[5].

If we, instead of jointly optimizing, greedily optimize the network parameters $\tilde{\boldsymbol{w}}$ followed by the GP hyperparameters $\tilde{\boldsymbol{\theta}}$ this approach was coined Deep Network for Global Optimization in [Snoek et al.] and was used for hyperparameter optimization. In our context we will be referring to it as Deep Neural Network with Bayesian Linear Regressor (DNNBLR).

2.4.3 Active Subspaces

A particularly simple form of manifolds are ones that are a linear transformation of the original space such that

$$f(\boldsymbol{x}) \approx g\left(\mathbf{W}^\top \boldsymbol{x}\right), \tag{2.53}$$

where the matrix $\mathbf{W} : \mathbb{R}^D \to \mathbb{R}^d$ projects $\boldsymbol{x}$ into a lower dimensional active subspace in which the *link function* $g(\cdot)$ will then take us to the response. Notice that $\mathbf{W}$ is a thin matrix so there exist multiple choices of column vectors that lead to the same subspace. So we are able to restrict ourselves to matrices $\mathbf{W}$ with orthogonal columns without loss of generality. In this way we can interpret the column vectors as the orthogonal directions in which the response surface varies the most.

Inspired by this observation a classical approach to discovering $\mathbf{W}$ finds a rotation of the space such that the direction in which the function varies the most are axis-aligned. To capture the notion of variability it starts by considering the gradient observations $\{\nabla f(\boldsymbol{x}_i)\}_{i=1}^N$. It then uses these samples to approximate the covariance between the gradients,

$$\mathbf{C} = \mathbb{E}_{\boldsymbol{x}}[\nabla f(\boldsymbol{x}) \nabla f(\boldsymbol{x})^\top] \approx \frac{1}{N} \sum \nabla f(\boldsymbol{x}) \nabla f(\boldsymbol{x})^\top. \tag{2.54}$$

Since $\mathbf{C}$ is a symmetric positive-definite matrix it allows for a eigendecomposition

$$\mathbf{C} = \mathbf{V} \boldsymbol{\Lambda} \mathbf{V}^\top, \tag{2.55}$$

[5]In practice we use the Blackbox Matrix-Matrix multiplication scheme for GPs [Gardner et al.]

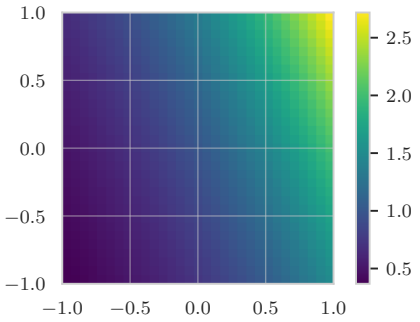


Fig. 2.3 An idealised toy example for AS is the function $f(x_1, x_2) = \exp(0.7x_1 + 0.3x_2)$. It varies along $[0.7, 0.3]$ but exhibits no change along $[0.3, -0.7]$. Using the gradient at sample points as an estimate AS finds the direction of most variability through SVD.

where the diagonal matrix $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_D)$ contains the eigenvalues of $\mathbf{C}$ in decreasing order and the columns of $\mathbf{V}$ are the corresponding orthogonal eigenvectors. Intuitively $\mathbf{V}$ rotates the input space so the direction of the maximal function variability are associated with largest eigenvalues. So to find the projection $\mathbf{W}$ into a lower dimensional space in which most of the action of $f(\cdot)$ is still captured we can consider the first biggest eigenvalues,

$$\mathbf{\Lambda} = \begin{bmatrix} \mathbf{\Lambda}_1 & 0 \\ 0 & \mathbf{\Lambda}_2 \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} \mathbf{V}_1 & \mathbf{V}_2 \end{bmatrix} \tag{2.56}$$

and pick $\mathbf{W} = \mathbf{V}_1$ as our $d \times D$ projection matrix. A toy example is shown in fig. 2.3.

We might not known d, in which case we have to determine how to split $\mathbf{\Lambda}$ and $\mathbf{V}$. If an active subspace exists, we should be able to detect a sharp drop in the spectrum of $\mathbf{C}$. In other words, we can go through the eigenvalues and cut-off at i when $\lambda_i \gg \lambda_{i+1}$.

In practice the eigenvectors and eigenvalues are found with Singular Value Decomposition (SVD). This puts a restriction on the applicability of the method as the classic implementations of SVD have $\mathcal{O}(N^3)$ time complexity. For very high dimensions, where N has to be large, this becomes unfeasible.

Active subspaces has been proposed in combination with Gaussian Processes with direct application in the financial setting in [Scheidegger and Bilionis] [Kubler and Scheidegger]. It proceeds by greedily learning the Active subspace and subsequently treating this as a feature space on which a GP is fitted. It is limited however by requiring gradients and being computationally expensive.

For a full treatment of Active Subspaces we refer to [Constantine].

2.5 Non-stationarity

We saw in section 2.2.1.1 that the covariance function encodes assumptions about the underlying function being modelled, such as its periodicity and smoothness. One commonly used kernel is the SE kernel because of its simplicity and its generic applicability across a whole range of problems. However, it encodes a strong assumption of infinitely differentiability which makes it ill-suited for functions where smoothness assumptions can be violated such as in the financial setting.

Solutions to this limitation can be split in two categories. One approach is to consider more expressive covariance functions such as combining kernels [Rasmussen and Williams, Duvenaud]. However, these are still limited by the covariance functions from which they are combined. The other approach instead pre-processes the data in such a way that a GP with a generic kernel is suitable for the transformed data. One such method considers a transformation of the output space [Snelson et al.]. Another type, which will be the subject of our focus, transforms the input instead. Several methods exists that learn this mapping prior to fitting the GP. In that case the choices are either guided by some heuristic or optimize an unsupervised objective. However, this greedy approach to determining the transformation might be suboptimal for the regression problem. Deep Kernel Learning [45] (or Manifold GP [Calandra et al.]) circumvent this problem by jointly learning the input transformation together with the hyperparameters of the kernel. This approach will conveniently also incorporate dimensionality reduction, thereby making it a suitable candidate for our setting.

It should be noted that the divide between expressive kernels and data transformation is a little artificial in that the input transformation can be absorbed into the kernel as we will soon see. In that way it can be considered as a more expressive covariance function.

2.5.1 Deep Kernel Learning

We have already described the method that jointly learns an input transformation with the GP hyperparameters. This was done in the context of dimensionality reduction however (see section 3.3.1).

The same approach is useful for learning non-stationarities. To see how recall that the latent space $\mathcal{L}$ acts as the input domain of the GP. Suppose we are modelling with a smooth GP prior such as the squared exponential. If our function f is not smooth over $\mathcal{L}$ the marginal likelihood $p(\boldsymbol{x}|\boldsymbol{\theta})$ which we are maximizing over $\boldsymbol{\theta}$ will be small.

Our parameterization of the mapping $\mathcal{X} \to \mathcal{L}$ will therefore be guided by this objective to find a transformation that leads to a smooth landscape.

Geometrically this allows for a stretching of the input space. The step function, in which the function is constant in intervals, provides a particularly illuminating example. Each interval can be mapped to a single point and spaced out such that a smooth kernel can be applied.

Again, as a particular case of DKL we will also be considering DNNBLR (section 2.4.2.2) for non-stationarity.

2.6 A Unified View of the Models

For understanding we include a unified overview of the different models. The models we consider can all be viewed as a combination of a feature extractor and a simple GP kernel with the exception of the inducing point method SGPR. This overview is provided in table 2.1. For reference we also provide time and space complexity more broadly for *all* models in consideration (see table 2.2).

First we note that a bayesian linear regressor turns out to be equivalent to a GP with a linear kernel [Rasmussen and Williams]. So DNNBLR is simply a neural network with a linear kernel. DKL simply allows for more expressiveness by considering *any* GP kernel. We restrict ourselves to stationary kernels and in particular the SE kernel to make it compatible with the scalable KISS-GP technique.

Recall that the main insight for SSGP was that a stationary kernel could be written as an inner product $\boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{x})$. This is exactly the definition of a linear kernel taken in the freqency domain which $\boldsymbol{\phi}$ maps to. So it is similar to DNNBLR except that it restricts itself to the frequency domain. Note however that the aim is different, in that it allows a fast approximation. We are *not* interested in learning a good feature mapping for expressiveness with SSGP (however when maximizing the spectral locations this will be a sideeffect).

This view is particularly nice when striving for a modular implementation of which is covered more generally in chapter 4.

Table 2.1 A unified view of the models.

	Feature extractor	GP kernel
BLR	Identity	Linear
SSGP	Frequency spectrum	Linear
DNNBLR	Neural network	Linear
DKL	Neural network	Any[1]
AS-GP	Linear	Any

[1] For scalability we are restricted to stationary product decomposable kernels such as the SE.

Table 2.2 Time and space complexity for training models given as a function of: the number of observations N, the number of inducing points M, the input dimesions D, and the dimensionality of the feature space d where applicable. The dashed line seperates the regression models from the purely dimensionality reduction methods.

	Time	Space
GP	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
GP Linear[1]	$\mathcal{O}(ND^2)$	$\mathcal{O}(N)$
DNNBLR[1]	$\mathcal{O}(Nd^2)$	$\mathcal{O}(N)$
SSGP	$\mathcal{O}(NM^2 + M^3)$	$\mathcal{O}(NM + M^2)$
SGPR	$\mathcal{O}(NM^2 + M^3)$	$\mathcal{O}(NM + M^2)$
KISS-GP$_{\text{(Toeplitz)}}$	$\mathcal{O}(N + M \log M)$	$\mathcal{O}(N + M)$
KISS-GP$_{\text{(Kronecker)}}$	$\mathcal{O}(N + DM^{1+1/D})$	$\mathcal{O}(N + DM^{2/D})$
AS	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$

[1] Specifically we implement it as a GP with linear kernel and exploit the results from [Gardner et al.].

Chapter 3

Experiments

We are ultimately interested in building a model which is well-suited for our economic settings. The theoretical descriptions and properties from chapter 2 still leave a big design space when it comes to concrete implementation. Take for instance Deep Kernel Learning, where the hyperparameters such as the network architecture strictly dictate the expressiveness and bias of the kernel.

This chapter aims to cover three experimental parts associated with our models. First we explore the properties of Deep Kernel Learning in particular. Secondly we compare our proposed models on synthetic embedding function both in other to compare the models but also to find suitable hyperparameters.[1] Finally, we explore the model on economic problems.

3.1 Setup

For reference we provide an exhaustive overview of model names and their default configurations as well as the error metrics used.

3.1.1 Methods

ExactGP/GP Gaussian Process using the Conjugate Gradient approach and Automatic Relevance Detection.

AS-GP Active Subspace with Gaussian Process (using CG iterations).

[1] By *hyperparameters* we do not refer to GP hyperparameters in this context but rather the parameters associated with the models such as network width and CG iterations.

KISS-GP Kernel interpolation for scalable structured Gaussian processes described in section 2.3.4.

SGPR The celebrated variational approach to inducing points [Titsias] covered in section 2.3.3.

SSGP Sparse Spectrum Gaussian Processes with optimized spectral locations from section 2.3.5.

SSGP fixed Sparse Spectrum Gaussian Processes with Monte Carlo sampled spectral locations from the kernel Fourier transformation.

DKL Deep Kernel Learning using CG iterations for the GP layer (described in section 2.4.1).

DKL KISS-GP Deep Kernel Learning using KISS-GP inducing points.

DNN A neural network using the same architecture as its DKL counterpart but with a final linear mapping to the output.

DNNBLR A neural network with a Bayesian linear regressor from section 2.4.2.2.

LASSO Least Absolute Shrinkage and Selection Operator, which we include as a baseline.

For both GP and DNN models we optimize the marginal log likelihood and mean square error loss respectively using the celebrated Adam optimizer which has been shown to be robust with respect to its hyperparameters for non-convex optimization problems. For a concise description of the method and a delightful read we refer to the original paper [Kingma and Ba].

We postfix DNN, DKL and DNNBLR with the neural network architecture using the layer notation described in section 2.4.1.2.

Environment The models are either run on an Intel Xeon E5-2660 CPU or Intel Xeon E5-2660 CPU. When GPU is enabled a NVIDIA GV100 Volta GPU 32GB is used. All experiments that compare performance are strictly run on the same environment.

3.1.2 Error Metrics

We are interested in doing well everywhere on the function we are estimating. To this end, we use the L_2 and L_∞ norm as our measure of error. This ensures that we penalize outliers—for L_∞ in an extreme way since it is the maximum. The Root Mean Square Error (RMSE) and maximum error (L_∞) are given as

$$\text{RMSE} = \sqrt{\frac{1}{T}\sum_{i=1}^{T}(f_i - \hat{\mu}(\boldsymbol{x}_i))^2} \qquad L_\infty = \max_{i \in \{1,\dots,T\}} |f_i - \hat{\mu}(\boldsymbol{x}_i)|, \tag{3.1}$$

where $\{ (f_i, \boldsymbol{x}_i) \}_{i=1}^{T}$ is the test set containing pairs of the true mean at associated input locations and $\hat{\mu}(\cdot)$ is our mean estimator.

However we are not only interested in the mean estimator but also in providing a reasonable uncertainty estimate. So we will also consider the Mean Negative Log Probability (MNLP) (see e.g. [Gelman et al.]):

$$\text{MNLP} = \frac{1}{2T}\sum_{i=1}^{T}\left(\left(\frac{f_i - \hat{\mu}(\boldsymbol{x}_i)}{\hat{\sigma}(\boldsymbol{x}_i)}\right)^2 + \log\hat{\sigma}(\boldsymbol{x}_i)^2 + \log 2\pi\right). \tag{3.2}$$

where we in addition introduce $\hat{\sigma}(\cdot)$ as the estimated standard deviation.

The errors are calculated using 2,500 uniformly random samples from the domain. In the case of synthetic functions we use the known mean, i.e. the noiseless variant.

3.2 Model Configuration

In this section we will investigate the behavior of the DKL model described in section 2.4.1. This will help us get an intuition for our model and ultimately guide how we configure both the network of DKL and DNNBLR as well as the training procedure which so far have been left unspecified.

3.2.1 Expressiveness of Deep Kernel Learning

3.2.1.1 Learning Varying Length-scale

We will showcase the capability for DKL to model functions with varying length-scales across the domain. These functions provide a slightly tricky objective for GPs with off-the-shelf kernels such as the Squared Exponential or Matérn as their single length-scale implies an assumption of a single frequency. When optimizing the hyperparameter by

maximizing the marginal likelihood this typically leads to learning a short length-scale. This is problematic for our generalization capabilities.

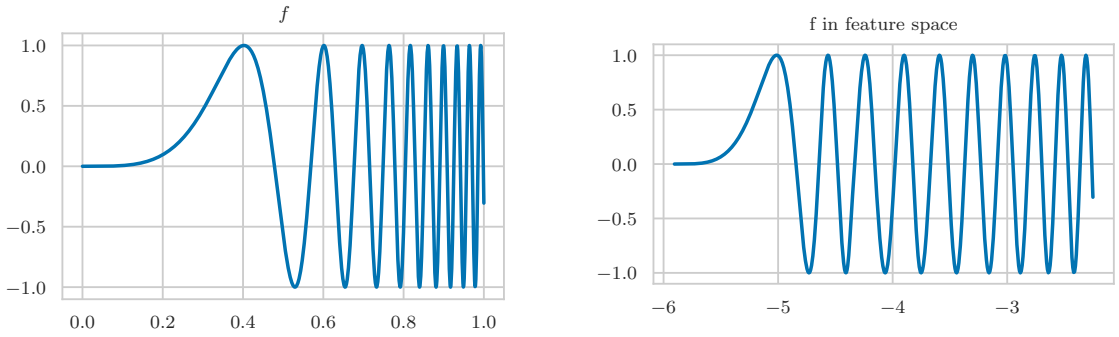
In [Calandra et al.] they illustrate how DKL can be used to learn an input mapping that concentrates the frequency spectra of these functions with otherwise wider spectrums. They argue that the capability to find these mappings is due to the joint training of the input transformation $\boldsymbol{\phi}$ and GP mapping g.

In a similar vein we consider the test function $f : \mathbb{R} \to \mathbb{R}$ with varying frequency across the domain

$$f(x) = \sin(60x^4).$$

The ability to learn an input transformation that makes the feature space stationary is illustrated in fig. 3.1. This will prove to be useful when moving to more complicated functions in section 3.3.

We observe that the RMSE is small even after less than 30 iterations of the Adam optimizer. Closer inspection shows that the feature mapping is close to an identity mapping. We assume that because of the coordinate-wise step size used by Adam it mostly takes steps in the direction of GP hyperparameters $\tilde{\boldsymbol{\theta}}$ since the gradient is much bigger compared to the single weights in neural network. If the neural network was not initialized to be something close to a identity map we might have observed $\boldsymbol{w}$ being adjusted earlier. Only after this initial convergence of $\tilde{\boldsymbol{\theta}}$ is the neural network adjusted to squeeze out the last bit of performance.



(a) (b)

Fig. 3.1 DKL can deal with a changing frequency by learning a suitable input transformation into a stationary feature space. For this particular example the DKL feature mapping [100-50-1] was used on samples from $f(x) = \sin(60x^4)$.

3.2.1.2 Learning Embeddings

Deep Kernel Learning is not only promising because of the ability to deal with non-stationarity but also as a means for dimensionality reduction. For illustrative purposes fig. 3.2 shows the deformation of the input space occurring when a *2-dimensional* feature space is learned. However, note that one of the dimensions can be integrated out without significant loss in accuracy. In section 3.3 a more thorough analysis is carried out where we also look at the effect of increasing the dimensionality.

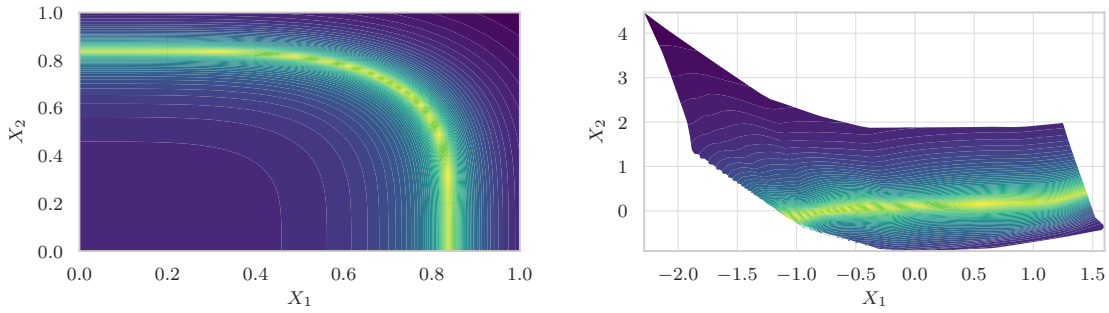


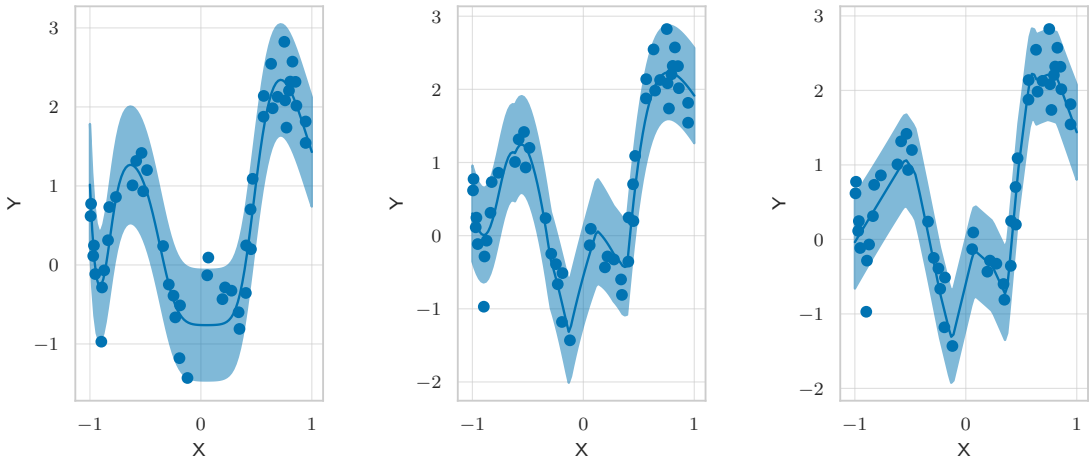
Fig. 3.2 Fitting DKL with [100-50-2] to an 1D function embedded in 2D space.

3.2.2 Neural Network Architecture

The neural network architecture in DKL and DNNBLR implicitly encodes our prior over functions. Of the many choices in constructing a network the activation functions have the biggest impact on the smoothness assumptions. A tanh activation function $\sigma_{\tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is simply a rescaled variant of the smooth logistic sigmoid function so our prior will favor smooth functions. The Rectified Linear Unit (ReLU) $\sigma_{\text{ReLU}}(x) = \max(0, x)$ on the other hand can more easily introduce kinks because of its discontinuity at zero. We therefore propose ReLU at every layer similar to what is done for DKL in [45]. This also has the added benefit of not suffering as much from vanishing gradients which leads to faster convergence.

In [Snoek et al.] they observed that using ReLU in DNNBLR at every hidden layer led to exploding posterior variance. We were *not able to replicate this phenomenon* and observe *contrary* to their example that ReLU provided reliable posterior variance both when used for DKL and DNNBLR. We believe this to be due to a particular implementation. In fig. 3.3 we illustrate with a toy-example the effect we observed for various combinations of activation functions.

[2]The same behaviour was observed using a DKL GP model.



(a) Tanh–Tanh (b) Tanh–ReLU (c) ReLU–ReLU

Fig. 3.3 This toy-example illustrates the effect of the activation function on the posterior. Each model is a DNNBLR^2 network fitted to the same 50 observations drawn from a GP with mean zero and a 0.1 length-scale SE kernel. A ReLU activation function at every layer including the last (denoted by ReLU–ReLU) captures possible discontinuities more easily. We did not observe the variance explosion that prevented [Snoek et al.] from using ReLU throughout the network even when testing across various depth, widths and epochs.

Recommendation Use ReLU for well-calibrated uncertainty estimates.

3.2.3 Greedy Training

A gradient step for DKL is asymptotically slower than for a neural network even when using inducing point methods. KISS-GP can make the asymptotics the same but in practice it still includes much bigger constants in it run-time. To this end, it is beneficial to consider if we can greedily train the neural network parameters separately, such that the joint training would require fewer iterations.

Conceptually, it is useful to note, that a GP can be viewed as a single-layer fully-connected neural network with infinitely many hidden units [Neal, Chapter 2]. So the full combined model including the feature mapping can be viewed as a single neural network.

So an intuitive approach is to replace this final infinite layer with a linear mapping to the output space and train it against the commonly used Mean Square Error loss. Then subsequently use those pre-trained weights as an initialization for the joint training. More precisely, we introduce the modified neural network

$$\bar{\boldsymbol{\phi}}_{\mathrm{NN}}(\mathbf{X}) = \mathbf{W}_{i+1}\boldsymbol{\phi}(\mathbf{X}) + \mathbf{B}_{i+1}. \tag{3.3}$$

The approach can then be summaries in the following two steps:

$$\begin{aligned} \text{greedy training}: &\quad \boldsymbol{w}' = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{N} (y_i - \bar{\boldsymbol{\phi}}_{\mathrm{NN}}(\boldsymbol{x}_i))^2 \\ \text{joint training}: &\quad \min_{\{\tilde{\boldsymbol{\theta}}, \boldsymbol{w}\}} p_{\tilde{\boldsymbol{\theta}}}(\boldsymbol{y}|\boldsymbol{w}) \quad \text{with Adam given initialization } \boldsymbol{w}'. \end{aligned} \tag{3.4}$$

One obvious problem with this approach is that the MSE loss might not be aligned with our objective of maximizing the log marginal likelihood. It is difficult to say anything in general, but for our problems we observed that we, regardless of this observation, could achieve smaller RMSE using the same computational resources if this greedy training was used. To gain further confidence we also note that this approach was successfully used to pre-train the neural network in [45].

The exact same greedy training strategy applies to DNNBLR. It is interesting to observe that the two losses in that case are better aligned.

3.3 High Dimensional Embeddings

In section 3.2 we showed some beneficial properties of DKL that suggest for what problems it will be useful. However it does not necessarily imply that it will perform significantly better than GP in those instances. We will now study DKL and DNNBLR on synthesized embeddings which allows us to compare it with well-understood AS-GP method. We are interested in understanding how the models should be configured for dimensionality reduction tasks.

In cases where an active subspace has been constructed we assume the effective dimensionality is known such that we can fix the dimensionality of the feature space on DKL. Note that we expect AS-GP to perform the best when the true object has been constructed to live in a lower dimensional linear subspace. What we hope is that DKL will still be able to recover in these cases.

We will first see how it provides a gradient-free approach to linear manifolds for which AS-GP is otherwise ideal. Then we will consider more general manifolds.

3.3.1 Linear Embeddings

We will consider the test function $f(\boldsymbol{x}) = \exp(\mathbf{A}\boldsymbol{x})$ with the choice of $\mathbf{A}$ illustrated in fig. 3.4 for dimensionality $D = 10$. We simply want to show that DKL and DNNBLR can capture the same low dimensional subspace as AS-GP consistently. Simultaneously it provides a good demonstration of using the code base to run several models multiple times across problems of different dimensionality.

We compare the following models: AS-GP, DKL, DNN, DNNBLR and LASSO. For DKL we choose an expressive network [1000-500-50-1] since we ultimately want to capture more complicated high-dimensional structures. DNNBLR and DNN uses the same [1000-500-50-1] network. For a more direct comparison with AS-GP we also include DKL [1]. All models uses quite aggressive L_2-regularization on weights of 0.01 except for [1] networks which is already restricted by the architecture. The same learning rate of 0.01 is used for all models. The number of epochs are chosen such that all models have approximately the same computational budget.

We observed that we needed roughly 20 times the dimensions to recover a feature space with low noise indicating that we had indeed learned a good approximation to $\mathbf{A}$. Without this requirement our expressive [1000-500-50-1] networks were incapable of learning a mapping that generalized for the regression task. This was slightly more than what Sneidegger et al. observed in [Scheidegger and Bilionis] was required for

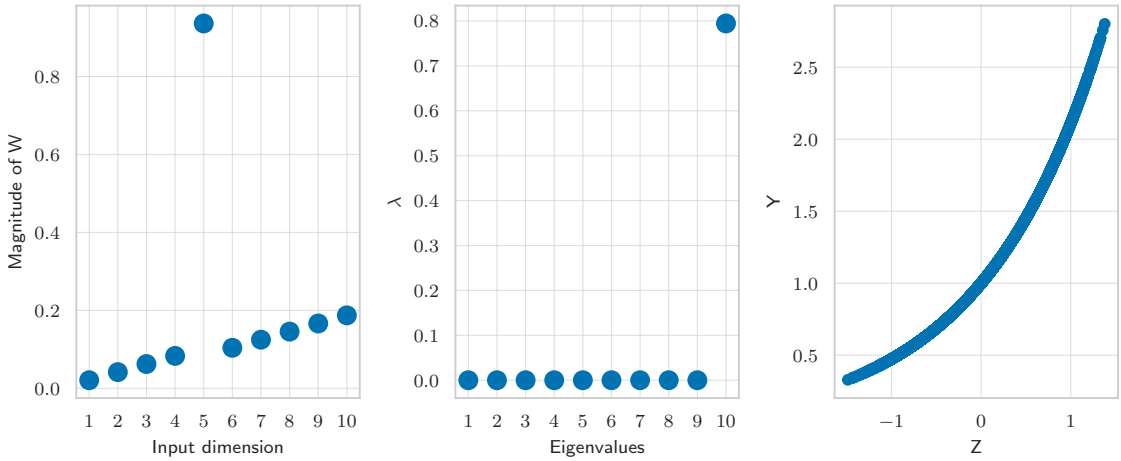


Fig. 3.4 An embedded exponential synthetic test function scalable in dimensionality D, $f(\boldsymbol{x}) = \exp(\mathbf{A}\boldsymbol{x})$. The *left-most* plot shows that one of the inputs dimensions are as active as the combined contribution of the other input dimensions. The *middle* plot shows that after a rotation based on the eigendecomposition of the gradients an Active Subspace of 1D is found. The *right-most* plot show samples from the response plotted in this 1-dimensional transformed space.

AS. Not surprisingly fewer samples than $20 \times D$ were required for the more biased and less expressive DKL network [1].

For 1000 dimensions this becomes computationally problematic since we require $N = 20,000$ observations. To this end we apply the GP approximation, SGPR, with a relatively small number of inducing points $M = 100$. This is possible since the big N is only required for the dimensionality reduction. The GP is fitted on a 1-dimensional subspace with an assumed large length-scale. This permits a small M if the inducing points are chosen carefully as is the case for SGPR.

Lessons from configuring hyperparameters Finding a suitable configuration of the neural network and learning parameters turned out to be difficult. We observed exploding loss for the joint training which further investigation showed was caused by exploding gradients. One type would periodically lead to increasing spikes in the loss. The second type would cause such a big step that the training would not be able to recover. To combat these symptoms we used two tricks: first, we clipped the gradient when the L_2-norm exceeded a certain threshold and, secondly, we used a type of early stopping described in Appendix B.4.

Finding the low dimensional subspace greedily proved essential—by which we mean greedily training the network with MSE loss as described in section 3.2.3. We

assume this is caused by a tricky hyperparameter landscape at the initialization when considering GP hyperparameters and network parameters jointly. It further allows us to speed up training. We comment on the problematic mismatch between the MSE loss and NLML in the discussion (section 5.1).

For greedy training to be effective we observed that some term that penalized model complexity was necessary now that the marginal log likelihood did not enforce this. A L_2-regularization of 0.01 on the weights proved to be ideal for these embeddings.

Greedy training did not mitigate spiky loss for non-trivial neural networks, however.[3] We list here several of the failed directions to fix it: batch normalization, adjusting the learning rate, choosing a shallower network such as [1000-1] and leaky-ReLU [Maas et al.].

Results Considering the amount of over-fitting likely caused by the fine-tuning the hyperparameter this experiment does not provide a completely fair comparison between models. However, we can still distill a few interesting points from the results in fig. 3.6.

Not surprisingly DKL [1] does well and better than DKL [1000-500-50-1] which shows that we can indeed come very close to AS-GP with a gradient free method for particular problem instances. DNNBLR does almost comparably with DKL followed by DNN. We observe that a pure DNN model can capture almost all the action–with more fine-tuning there is no doubt that we should be able to match the performance of DKL. This observation is problematic for uncertainty quantification which we will cover a little further down.

Even with fine-tuned hyperparameter we do not seem to have found an optimal configuration for our computational budget. To see why, notice that by construction we should get lower error for big D as in the case of AS-GP. However, the neural network based implementations, DNN, DKL and DNNBLR, does not improve for increasing D seemingly meeting a bottleneck. We believe this bottleneck is caused by the learning parameters: too small gradient steps combined with gradient clipping prevents it from converging to a local minimum. Finding a configuration that consistently converges within a reasonable budget is still not answered fully.

However, even though the results does not seem promising, closer inspection shows that we have indeed learned the mapping. We depict the prediction in the feature space constructed by [1000-500-50-1] DKL and DNNBLR for the 1000D embedding in fig. 3.7. Neither predictions deteriorate too much–that is relative to the output space their error is still small. The same observation can be made in the noisy setting for $\epsilon = 0.01$

[3] By *non-trivial* we exempt the linear mapping [1].

which we have included in section C.0.1 (note that the comparison with AS-GP is not quite fair as the model is provided access to noise-free gradient observations). It suggests that these expressive models can replace AS-GP.

The MNLP (see section C.0.1) follows the RMSE which is not surprising considering the 1D subspace is densely and approximately uniformly covered in samples. This leads to a problem for uncertainty quantification: The neural network $\phi(\cdot)$ might learn the correct mapping $\mathbf{A}$ to the subspace which allows the GP part of DKL to learn the minimal possible noise-prior. In itself this is not problematic, but consider if a mapping, $\bar{\mathbf{A}}$, is learned which only correctly projects the training data. Then DKL will be overconfident for test predictions so without sufficiently regularizing DKL our uncertainty estimates becomes uninformative.

The optimistic result is that we can indeed get reasonable results for DKL in AS-GP regimes *if the network is regularised sufficiently*. Further, DKL achieves this without gradients. We note that it is also very appealing to consider the simpler DNNBLR model which seems to perform comparatively to DKL for large D.

So even though DKL comes at a cost of accuracy for linear embeddings, it still provides meaningful estimates. In section 3.5.2 this allows us to achieve convergence in an economic growth model. Combined with scalable methods such as KISS-GP this could potentially handle unprecedented dimensionality in the economic setting.

Hyperparameter optimization: a futile attempt After discovering the important hyperparameters we tried settling on a configuration for our experiments by searching the hyperparameter space. Specifically, instead of using sequential methods such as Bayesian Optimization (BO), we chose 500 points uniformly at random from our hyperparameter space. The reason was impatience, as this allowed us to evaluate the RMSE for each of our configurations in parallel using our HPC workflow (see chapter 4). We deviated from naive random search by fitting a GP to these evaluations and maximizing its posterior mean to obtain the final configuration.[4] As a final detail we let us be inspired by the Bayesian Quadrature literature by transforming the output using the logarithm before training [Osborne et al.]. This ensured that we did not make invalid negative predictions of the RMSE. However, the posterior turned out to be too multi-modal to provide any useful singular proposal. Instead, we chose (partially guided by this posterior) to pick the following: learning rate of 0.001, relatively large greedy training of $3,000$ epochs, and reasonably aggressive gradient

[4]For the reader familiar with BO this is equivalent to full exploitation in the BO setting.

(a) Samples $\{ \boldsymbol{x}_i \}_{i=1}^N$ in the hyperparameter space $\mathcal{X}$.

learning rate: $x_i^{(1)} \in [0.001, 0.1]$
greedy epochs: $x_i^{(2)} \in [300, 10000]$
epochs: $x_i^{(3)} \in [300, 3000]$
weight decay: $x_i^{(4)} \in [10^{-6}, 10^{-2}]$
gradient clipping: $x_i^{(5)} \in [1, 1000]$

(b) The hyperparameter optimization steps.

1. Uniformly sample on a logarithmic scale $\tilde{\boldsymbol{x}}_i \in \log_{10} \mathcal{X}$.
2. Transform all samples into the hyperparameter space, $\boldsymbol{x}_i = 10^{\tilde{\boldsymbol{x}}_i}$.
3. Evaluate a model for each $\boldsymbol{x}_i$ in parallel to obtain the RMSE denoted y_i.
4. Fit a GP to $\{ (\tilde{\boldsymbol{x}}_i, \tilde{y}_i) \}_{i=1}^N$ where $\tilde{y}_i = \log_{10} y_i$.
5. Minimize the GP posterior to obtain the optimal $\tilde{\boldsymbol{x}}$.

Fig. 3.5 The hyperparameter optimization scheme leveraging HPC. We are a little loose on notation for brevity. Greedy epochs refer to greedy training as described in section 3.2.3.

clipping and weight regularization of 2 and 0.01 respectively. We include the procedure in fig. 3.5, nonetheless, as it might be of interest.

3.3.2 Non-linear Embeddings

Expressive models such as DKL and DNNBLR carry the promise of learning the more general non-linear embeddings. This is in contrast to AS-GP which breaks down if the linear assumption does not hold, as illustrated in fig. 3.8. The promise of generalization turns out to be difficult in practice, however.

To investigate this point we tested the models on $f(\boldsymbol{x}) = g(\sum_i^D x_i^2)$ for various choices of g using the [1000-500-50-1] architecture as for linear embeddings in section 3.3.1. We were not able to consistently recover the subspace in higher dimensions. For illustrative purposes we include the results in table 3.1 for the Sinc function $\text{sinc}(x) = \frac{\sin(x)}{x}$ embedded in 4 dimensions using 5,000 training samples. We observe that DKL is capable of learning the non-linear subspace, which is also apparent from the error metrics.

In itself a low dimensional setting such as 4D is not particularly interesting as dimensionality reduction is not strictly necessary. However, being able to capture the manifold suggests its use in higher dimensions.

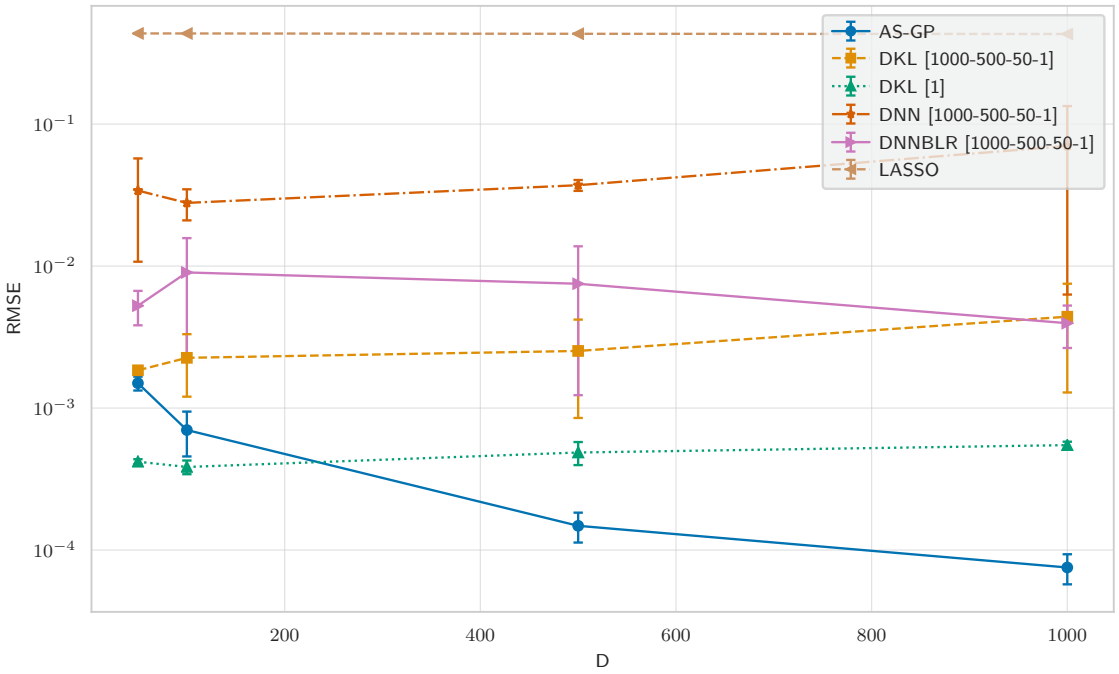
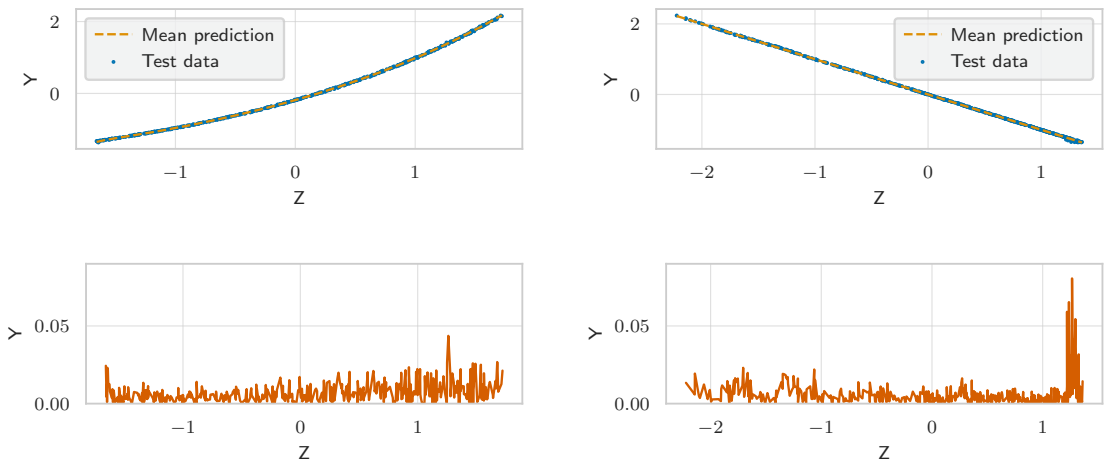


Fig. 3.6 RMSE performance on increasing D on the linear embedding $\exp(\mathbf{A}\boldsymbol{x})$.



(a) DKL [1000-500-50-1]

(b) DNNBLR

Fig. 3.7 Plots of the predictions of DKL and DNNBLR in the feature space Z in a noise-free setting. The posterior variance has been left out for clarity as it is negligible. Both models learn a reasonable approximation of the active subspace–similarly in the noisy setting. The bottom plots shows the absolute error. (Be aware that the linear interpolation plotted is not an accurate depiction of reality.)

$$f(\boldsymbol{x}) = \frac{1}{|0.5 - \Phi(\boldsymbol{x})| + 0.1}$$
$$\Phi(\boldsymbol{x}) = \sum_i x_i^2 \quad \text{where} \quad \boldsymbol{x} \in [0,1]^D$$

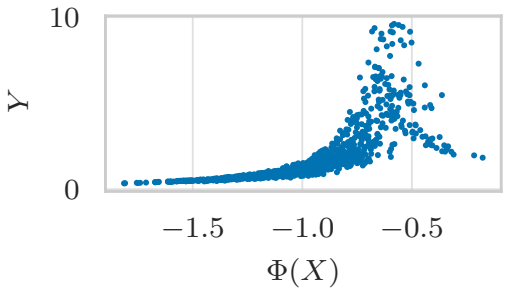


Fig. 3.8 The active subspace of a noise-free kink embedded in a 3D hypersphere (The eigenvalue for the 1D active subspace is 6 times higher than the remaining eigenvalues). Observe that the noise level of the function in this space is intrinsically higher. Thus we cannot hope to estimate the true function values well only using observations in this space.

Table 3.1 We ran AS-GP, DKL and DNNBLR on a 4D non-linear embedding of the Sinc function $f(\boldsymbol{x}) = \text{sinc}(\sum_i^D x_i^2)$. DKL and DNNBLR use a [1000-500-50-1] architecture. All models are trained on 5,000 samples and statistics are calculated over 5 independent runs. Closer inspections of the feature space shows that only DKL manages to find the subspace while DNNBLR only finds a good approximation.

Model	RMSE	L_∞	MNLP
AS-GP	0.036 ± 0.000	0.21 ± 0.00	-0.7 ± 0.0
DKL	0.007 ± 0.003	0.13 ± 0.05	-1.4 ± 0.1
DNNBLR	0.013 ± 0.004	0.15 ± 0.01	-1.2 ± 0.1

3.4 Comparison with Many Datapoints

In this section we will study different scalable GP approaches in isolation. This is motivated by the curse of dimensionality in section 3.3.1 in which we needed many data points to detect the structure of the manifold. In the case of DKL, where the feature mapping and GP is fitted jointly, the most trivial solution is to make the whole model scale. To this end we consider the inducing points methods and interpolation points method KISS-GP.

3.4.1 Influence of the Number of Inducing points

Undeniably an inducing point method without kernel interpolation will do better since it provides an *approximation* after all.[5] However, since KISS-GP scales asymptotically more slowly in M we assume we can achieve better error for the same run-time.

Following [Wilson and Nickisch] we consider the Natural Sound dataset original introduced by [Turner] in a different context. The objective is to model a 1D natural sound time series such as to recreate large contiguous missing regions. The dataset consists of 59,306 training samples and 691 set samples which puts it outside what is feasible to model with exact GP. It satisfies one of the sufficient criteria from section 2.3 for being a statistically challenging problem as it has high variability.

The models we will be comparing are the inducing point method SGPR from section 2.3.3, the sparse spectrum SSGP (section 2.3.5), and, finally, the inducing grid method KISS-GP (section 2.3.4). Toeplitz structure of the inducing points is exploited for KISS-GP since we are in 1D. For SSGP we considered both the variant that optimizes the spectral points and the one that keeps them fixed. However, since there was no significant difference between the two for this particular data set, we have only kept SSGP-FIXED to avoid clutter.

We are ultimately interested in comparing the models wall-time. To ensure a fair comparison we also use the CG iteration for SSGP and SGPR instead of restricting this implementation to KISS-GP. This ensures that the wall-time differs not because of implementation details (such as optimization steps etc.) but rather by the influence of M alone.

Figure 3.9 shows how M influences the error with fixed N—in this case using all 59,306 training examples. Notice how the RMSE for SGPR drops faster than that of

[5] Note that strictly speaking an approximation can in some instances do better as it is undeniably biased in some way.

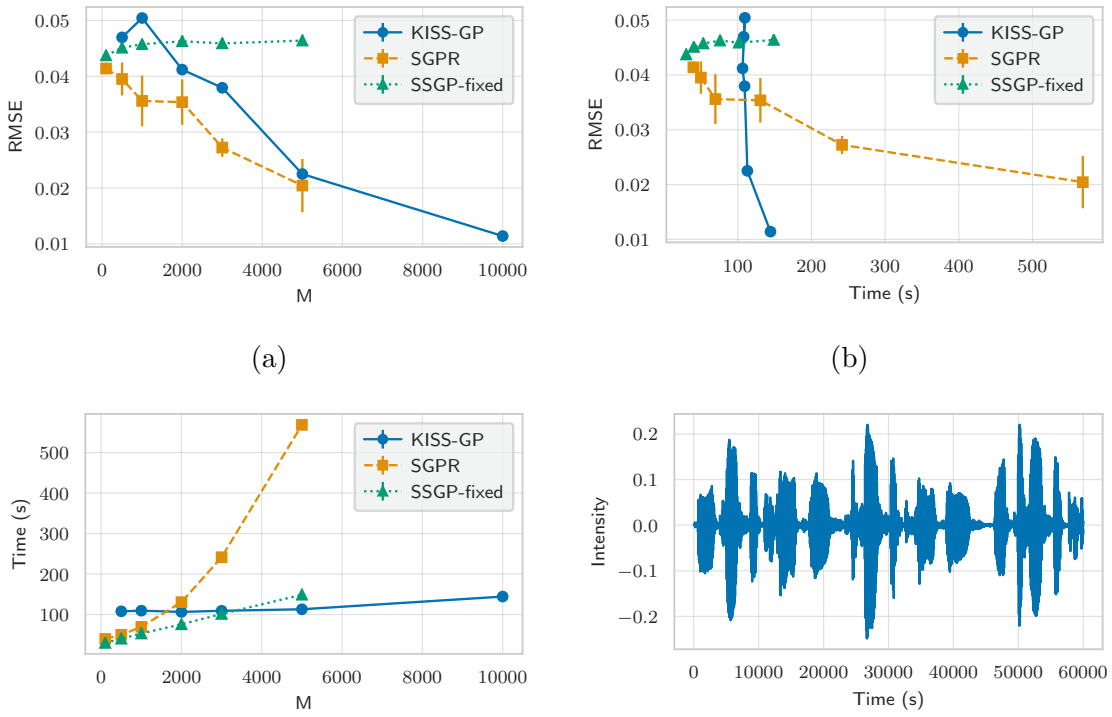


Fig. 3.9 Modelling Natural Sound with three different inducing point methods KISS-GP, SSGP and SGPR. It shows the influence of the number of inducing points for the RMSE and the run-time for each model. For each model we plot the mean with the standard division as error bars taken across 5 executions.

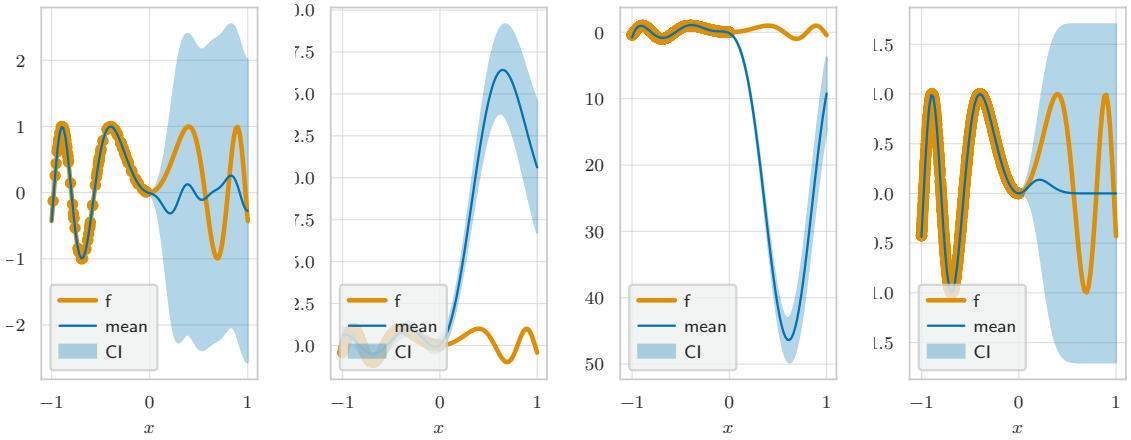
KISS-GP when considered as a function of M in fig. 3.9a. This is to be expected as the locations of the inducing points are optimized in SGPR.

The advantage of KISS-GP only shows when viewing RMSE as a function of wall-time. This is because the running time scales much more slowly in M for KISS-GP. In particular we observe that KISS-GP with 10,000 inducing point has approximately the same runtime as SGPR with 500 inducing points (fig. 3.9c). In fig. 3.9b we see how this allows us to achieve lower RMSE by increasing M with effectively no effect on the wall-time.

Concerning SSGP it is interesting to note that the model fails to predict the missing regions on the natural sound dataset. Not only does it not improve with increasing M it further does not achieve a better RMSE than a mean estimator would. We expect this to be a particular case of the known phenomenon of *variance starvation* observed in [44]. They observe that both mean and variance estimate of SSGP in regions with unobserved data where bad. That is, the mean was far away from the prior mean with relatively high confidence (i.e. the variance stays unaffected). This was an increasing problem as the training size became larger. In fig. 3.10 we recreate this problem for a 1D sample draw of a GP. This renders SSGP inappropriate for this particular experiment even though the running time is still informative.

This turns out to be a particularly nice case for KISS-GP. The signal has a very low length-scale which makes it necessary to cover the entire domain with many observations. Said in another way, a GP using only a subset of observations has significant performance drop. Similarly, for inducing point methods such as SGPR the relatively few inducing points in the input domain restricts how informative the abundant number of observations can be. KISS-GP on the other hand permits more inducing point, allowing us to capture the structure of signal. In [46] they make exactly this case by showing that normal inducing points methods are most suitable for smoothing kernels and fail when trying to learn more expressive kernels. This is especially problematic for the economic setting in which we know a more expressive kernel such as the DKL is required.

Regardless of KISS-GP's superior performance, SGPR still manages to perform reasonably well—it is primarily restricted by the computational complexity associated with the number of inducing points M. However, the KISS-GP approximations is not a silver bullets. We will later, in section 3.5.1, see an economic example where SGPR champions and does so regardless of its worse asymptotics.



(a) SSGP $N = 100$ (b) SSGP $N = 500$ (c) SSGP $N = 2000$ (d) GP $N = 2000$

Fig. 3.10 Variance starvation of SSGP in which the posterior in regions with no observation becomes uninformative as N increases. We use $M = 250$ inducing points for SSGP and observe that it produces an unreliable confidence interval for the mean as N becomes larger than M. This is a problem which exactGP does not face as illustrated in d. We note that we only observe this misbehavior when learning the hyperparameters. That is, for fixed hyperparameters, the predictions across varying N are similar.

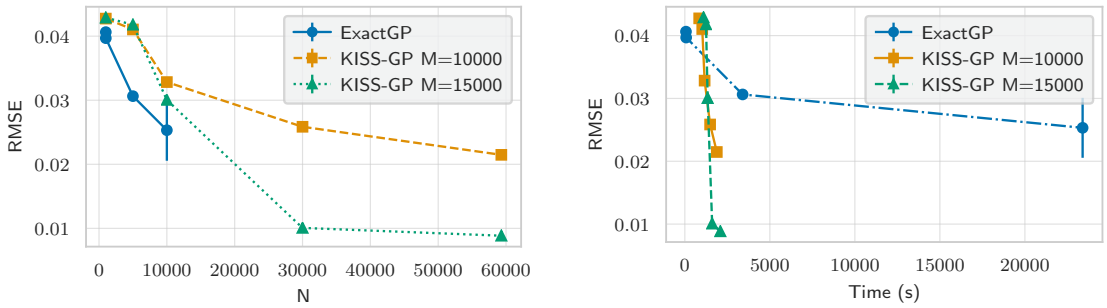
3.4.2 Improving by increasing number of observations

We cannot hope to do better than an ExactGP. However, we might be able to scale to observations previously impossible and thereby achieve better predictive ability than an exact GP trained on less data.

We use the same natural sound dataset and prediction objective described in section 3.4.1 but now use it to study the influence of N. We compare the trend of ExactGP (until it is computationally infeasible) with the scalable KISS-GP. In the experiment we keep a high, fixed number of inducing points M to prevent it from being the bottleneck of the estimate.[6].Specifically we pick $M = 10,000$ as this yielded a reasonable run-time in section 3.4.1 in which all $59,306$ samples where used. In addition we include $M = 15,000$ to see the interaction between M and N.

We run both KISS-GP and ExactGP for increasing N until the maximum of $N = 59,306$ is reached or our computational budget (given as a time constraint) has been exceeded. Results are shown in fig. 3.11.

[6]However note that convergence issues for CG are even more profound when M is large. Increasing M might give better estimates in theory but we run into the danger of numerical instability mentioned in Appendix B.6.



(a) (b)

Fig. 3.11 We show with the natural sound dataset that KISS-GP can outperform an ExactGP by scaling to more datapoints. Note that all executions are done on a CPU, thus the time discrepancy with fig. 3.9.

This experiment shows that it is possible to regain the lost accuracy caused by the approximating made by both the inducing points and the kernel interpolation used in KISS-GP. For small N ExactGP outperforms KISS-GP, as expected, but as N keeps increasing we achieve the same accuracy with KISS-GP (albeit using a larger number of observations). We observe that ExactGP has the same error for $N = 10,000$ as KISS-GP has for $N = 30,000$ when using $M = 10,000$ inducing points.

For even larger N The accuracy is even improved as the faster asymptotic running time allows for many more observations effectively compensating for the lost accuracy. This use of large N is permissable as the running time comparison in fig. 3.11b shows that the computational effort is still much lower than ExactGP using $N = 10,000$.

A few details remain. It should be noted that the KISS-GP is here compared with ExactGP using double precision. This higher precision turned out to be *necessary* for ExactGP to achieve good performance whereas KISS-GP was run using floating points (allowing for faster computations). It has been shown that CG iterations using floating points can in some cases beat Cholesky-based approaches even when these use double precision [Gardner et al., Figure 1]. This does not explain this phenomenon however, as ExactGP was performant using CG as well. Conversely, why KISS-GP does not suffer when using floating points remains unexplained.

We also tested KISS-GP for $M = 20,000$ but discovered that the performance was declining. We assume this is because enough CG iterations were not allowed as the problem becomes harder for larger N and M.

Recommendation In more generality, we have observed that KISS-GP only becomes effective when we have inducing points $M > 10,000$. Regardless of the slow asymptotic scaling this still renders the method undesirable for small N. We therefore suggest that KISS-GP is used only when $N > 10,000$.

3.5 Economic Setting

We will now turn to the economic applications and as a proof-of-concept show the models on a range of problems. We will model policy functions and embed the model into value iteration to solve a dynamic optimal control problem.

Instead of settling on some choice of hyperparameters for each of our models ex-ante as would be the usual approach, we allow ourselves to compare multiple variants of each model. This is because our goal is not primarily to demonstrate the robustness of our models. Rather, we still want to understand our models better and investigate what works for the economic setting specifically. In some sense we are indeed interested in over-fitting to this domain!

3.5.1 Heston Option Pricing

To test SGPR, SSGP and KISS-GP in isolation on a economic application we consider the option pricing using the Heston model (see fig. 3.12). The model is a simulation which makes it a useful case study as it can be systematically scaled to arbitrary observation numbers and locations. Another key point is that it allows for exact evaluation of performance since we know the ground truth.

For KISS-GP we set the number of inducing points equal to the number of training points (i.e. $M = N$) with a maximum of $10,000$. These inducing points are placed on a 2-dimensional cartesian grid so we exploit both Kronecker and Toeplitz structure to achieve scalability. For SSGP, SSGP-FIXED and SGPR we use $M = 2,048$ inducing points. For the SSGP models this implies $1,024$ spectral points which are represented with a cosine-sine pair. All models are run on a GPU with results illustrated in table 3.2.

Contrary to the case in section 3.4, SSGP and SGPR perform well. We attribute this to the response surface having much longer length-scale. This also allows us to compare SSGP and SSGP-fixed. For this particular case SSGP with its optimized spectral points performs better than the Monte Carlo based SSGP-fixed method. Overall, SSGP methods perform worse than the other methods. Not surprisingly, SGPR and KISS-GP

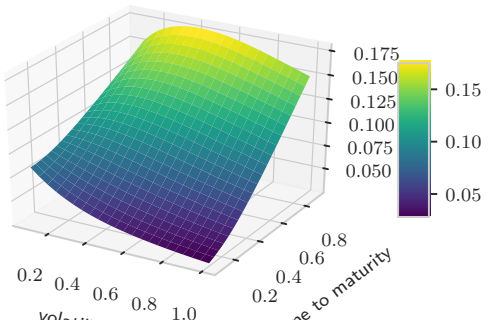


Fig. 3.12 A dense generated grid for the Heston based option pricing. Each evaluation is evaluated by simulation.

perform comparably with exact GP, considering the high number of inducing points used. Noteworthy is the fact that they seem to extrapolate the performance trends of exact GP for bigger N. KISS-GP, however, does not gain any advantage over SGPR by seeing more observations.

The most interesting observation from this study, however, is probably that *exact* GP is remarkably fast when GPU-accelerated. When speeding up the CG iterations on a GPU we can train a GP with 20,000 points in 160s and predict 2,500 points in less than a second.[7] However, at 40,0000 observations the bottleneck becomes memory by requiring more than what could fit on a 32GB RAM GPU. This is followed by SGPR and SSGP's failure at $N \geq 250,000$. We comment further on this in the discussion (chapter 5).

Remarks In the initial experiments we observed degrading performance as N was increased even for exact GP albeit using the CG iteration method. To avoid this we set the maximum number of CG iterations and the preconditioner size large for all experiments and relied on the tolerance threshold to stop early. For large N, having a large preconditioner size led to much faster convergence. We note that this makes a time comparison unfair.

Initial runs of Cholesky based GPs showed that CG with floating point was competitive with Cholesky using double precision and required significantly less time. This is consistent with the findings in [43]. Note however that this required a preconditioner of sufficient size–effectively it becomes a space-time tradeoff.

[7] We are leveraging the implementation of [Gardner et al.].

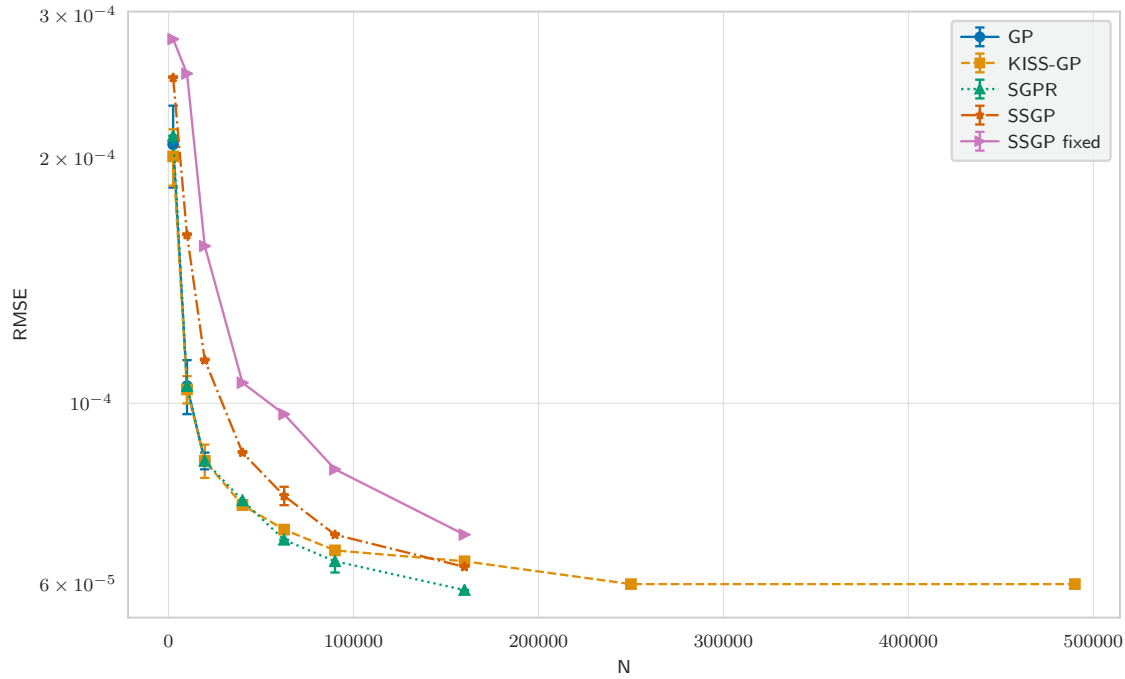


Table 3.2 Heston based option pricing results. Inducing point are set equivalent to N for KISS-GP but capped at $10,000$. For SGPR and SSGP $M = 2,048$ inducing points are used. For SSGP this means $1,024$ spectral points, each represented by a cosine/sine pair. GP exceeds the GPU memory limit for $N \geq 40,000$ while SSGP and SGPR fail at $N \geq 250,000$. SGPR and SSGP do well in this reasonably smooth setting where the limited number of inducing points is not restrictive.

3.5.2 Stochastic Optimal Growth Model

[Scheidegger and Bilionis] showed that AS-GP scales to previously impossible problem sizes when considering economic growth models which have a small active subspace. The method was however limited by being computationally infeasible for dimensions higher than 500 since an order of 10^5 observations would be required to learn the subspace which is prohibitively large for both AS and GP. We will now show that DKL-KISS-GP has the potential to scale the model to even higher dimensions, namely 1000. This is possible since the model which learns the feature mapping and the GP jointly scales linearly in the number of observations.

To illustrate this we will use the growth model described in [Cai and Judd] which has become one of the standard test-beds when studying methods for economic models in high dimensions. On a high level the model is an infinite-horizon, discrete-time and multi-dimensional stochastic model (a generalized description can be found in section 2.1). It is particularly useful for our study since it allows us to scale up the dimensionality of the problem in a meaningful way as the dimensionality is proportional to the number of sectors involved.

In section 3.3.1 we saw that DKL was able to learn the active subspace in synthesized datasets when fixing the feature space dimensionality d. For the growth model we use the observation made in [Scheidegger and Bilionis] that the subspace dimensionality is 1 and we will assume this is known for the model so it can be exploited.

3.5.2.1 Method

To find a solution to the optimal growth model we embed a GP into the value iteration described in section 2.1 similarly to [Scheidegger and Bilionis]. More precisely, we model the value function at every iteration with a GP fitted to N observations. These observations $\{ (\boldsymbol{x}_j, y_j) \}_{j=1}^N$ are exact evaluations of eq. (2.3), i.e. $y = V_i(\boldsymbol{x})$.

The two main computational bottlenecks comprise of fitting the GP and evaluating eq. (2.3). As our GP, we use a DKL KISS-GP model which handles the former. Note that the dimensionality reduction of DKL is crucial here as KISS-GP scales exponentially with the dimension. For the latter, observe that at each iteration the evaluations are *embarrassingly parallel* as they have no inter-dependencies. So between every iteration we distribute $\{ \boldsymbol{x}_j \}_{j=1}^N$ evenly across our cluster nodes to obtain $\{ y_j \}_{j=1}^N$.

3.5.2.2 Results

In fig. 3.13 we illustrate value iteration applied to the growth model using GP and DKL KISS-GP as models for the value function. In this initial study we only consider a 50 dimensional case due to constraints on computational resources. The plot provides a way to show convergence of the value iteration by showing the difference between a value function and the preceding value function. For 50 dimensions a scalable approach such as KISS-GP is not strictly required. Nonetheless, the experiment was carried out for DKL KISS-GP to test the robustness.

The experiment required approximately 2,000 node hours. Only by distributing it across 30 nodes using Message Passing Interface (MPI) were we able to get a reasonable running time of less than 3 days. Note that this is still far from what [Scheidegger and Bilionis] achieved with their highly optimized code running a 500 dimensional model in 3,700 node hours. However, it is sufficient to verify the applicability of the model.

There are two concerns when applying DKL KISS-GP to value iteration: that either the active subspace is not found or the KISS-GP approximation is too aggressive—both potentially preventing convergence. Figure 3.13 shows that it indeed converges. Further, it does so with a rate similar to that of GP.

This proof-of-concept shows that DKL KISS-GP can reliably be injected into a value iteration scheme. Combined with the observation made in section 3.3.1 that DKL can learn high-dimensional embeddings this suggests that the method could scale to even 1,000 dimensions. Additionally, in contrast to AS-GP it would do so without requiring gradients and generalize to more complicated economic models in which the low dimensional manifold is not necessarily assumed linear.

3.5.3 High-dimensional Policy Function

To investigate the expressive kernel DKL in the economic setting we turn to policy functions with discontinuities. Specifically we will consider the international real business cycle (IRBC) model studied in [Brumm and Scheidegger] where investment in each country is irreversible.[8] This assumption is crucial for our studies as it induces kinks in our policy function. Similar to the growth model in section 3.5.2 we can scale the dimensionality of the problem meaningsfully—here by increasing the number of

[8]For a full discription of the model we refer to [Juillard and Villemot] and to [Brumm and Scheidegger] for the investment irreversible assumption.

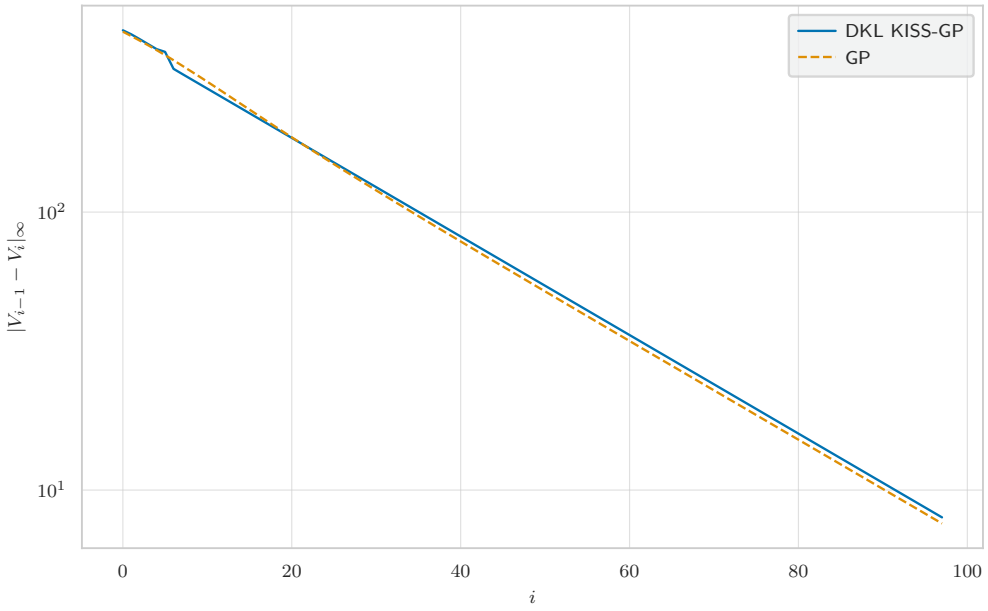


Fig. 3.13 Value function convergence for a 50-dimensional optimal growth model showing the decreasing L_∞ error $||V_{i-1} - V_i|_\infty$.

countries. We will specifically be considering policy functions with dimensionality ranging from 2 to 20 (an example in 2D is illustrated in fig. 3.14).

A solution is found by a recursive time iteration algorithm in which A-SG is applied at every step. We will however restrict ourselves to modelling the final policy obtained by the algorithm. This will allow us to more directly inspect the models.

We compare DKL [1000-500-50-2] with GPs using both a Matérn and an SE kernel. Since DKL also permits the more expressive Matérn kernel when scalability is not a concern, we include it for completeness. This will allow us to better understand the effect of the neural network. To understand the GP part we also include DNNBLR [1000-500-50-2] and DNN [1000-500-50-2]. As a baseline a standard least absolute shrinkage and selection operator (LASSO) is used. The L_2 and L_∞ error with confidence interval are provided for all models in table 3.3 with MNLP results included in section C.0.2.

Notice that SE clearly makes too strong a smoothness assumption. Not surprisingly Matérn performs better across the board. Interestingly DKL (Matern) and even DKL (SE) are comparable across all error metrics. At first glance it is not so remarkable that a Matérn kernel *with an expressive input mapping* does just as well. What *is* surprising is that it manages to do so *despite* the dimensionality reduction for $D > 2$.

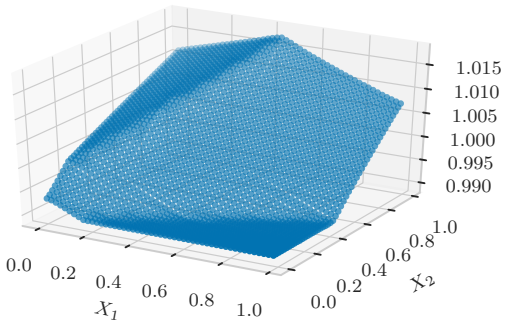


Fig. 3.14 Policy function in 2 dimensions.

Knowing that the Matérn kernel does not provide an accurate prior for our problem we could still hope that the expressive DKL kernel could provide better predictions. There are two reason for DKL to fail: either it does not converge because it is not given enough computational resources or it is stuck in a local optimum since the high-dimensional optimization problem is harder.

The interpretation of the MNLP results found in section C.0.2 tells a slightly different story. Overall, the MNLP for vanilla GP is better than DKL and DNNBLR which we attribute to a more accurate uncertainty estimate. Interestingly DNNBLR also has better MNLP than DKL. However, both models have high variance of the MNLP when computed over 5 independent runs. Since this is *not* the case for RMSE it must be due to varying posterior variance. Some way of stabilizing this seems worthy of investigation.

			RMSE		
Dimensionality	2	4	8	12	20
Model					
GP (Matern)	0.00002 ± 0.00000	0.00105 ± 0.00002	0.00102 ± 0.00000	0.00117 ± 0.00000	0.00074 ± 0.00000
GP (SE)	0.00008 ± 0.00000	0.00310 ± 0.00063	0.00144 ± 0.00010	0.00167 ± 0.00009	0.00097 ± 0.00003
DKL (Matern)	0.00003 ± 0.00000	0.00124 ± 0.00013	0.00130 ± 0.00007	0.00112 ± 0.00005	0.00076 ± 0.00004
DKL (SE)	0.00003 ± 0.00000	0.00128 ± 0.00006	0.00123 ± 0.00003	0.00107 ± 0.00005	0.00082 ± 0.00006
DNNBLR	0.00012 ± 0.00004	0.00113 ± 0.00009	0.00133 ± 0.00019	0.00119 ± 0.00010	0.00178 ± 0.00022
DNN	0.00063 ± 0.00057	0.00979 ± 0.01051	0.01524 ± 0.01737	0.00669 ± 0.00277	0.01850 ± 0.03356
LASSO	0.00805 ± 0.00000	0.10609 ± 0.00000	0.10138 ± 0.00000	0.10534 ± 0.00000	0.08017 ± 0.00000
			L_∞		
Dimensionality	2	4	8	12	20
Model					
GP (Matern)	0.00008 ± 0.00000	0.00239 ± 0.00005	0.00372 ± 0.00002	0.00364 ± 0.00005	0.00242 ± 0.00006
GP (SE)	0.00035 ± 0.00001	0.00973 ± 0.00263	0.00535 ± 0.00042	0.00657 ± 0.00072	0.00343 ± 0.00006
DKL (Matern)	0.00017 ± 0.00003	0.00412 ± 0.00127	0.00423 ± 0.00041	0.00341 ± 0.00044	0.00321 ± 0.00068
DKL (SE)	0.00021 ± 0.00002	0.00396 ± 0.00077	0.00376 ± 0.00040	0.00317 ± 0.00046	0.00317 ± 0.00050
DNNBLR	0.00046 ± 0.00009	0.00360 ± 0.00037	0.00404 ± 0.00072	0.00337 ± 0.00054	0.00562 ± 0.00123
DNN	0.00191 ± 0.00154	0.02000 ± 0.02517	0.03245 ± 0.03648	0.01694 ± 0.00536	0.04629 ± 0.08508
LASSO	0.01562 ± 0.00000	0.19826 ± 0.00000	0.19644 ± 0.00000	0.19869 ± 0.00000	0.14742 ± 0.00000

Table 3.3 To study, in particular, DKL's capability to handle kinks, we fit our models to policy functions of dimensionality $D = [2, 20]$ from the IRBC model. The results for all experiments show mean ± one standard deviation computed over at least 5 runs.

Chapter 4

Implementation

A crucial part of our contribution is the implementation, which has allowed us to run experiments frictionlessly and get a confidence interval for all of our results.

From a user perspective the experience can roughly be divided into two workflows: 1) the local workflow for investigating a model on our machine, and 2) a server workflow permitting many experiments to be run in parallel on a cluster.

The chapter is organized as follows: first we motivate the need for reproducable configuration of models in section 4.1, we then cover the two model execution flows in section 4.2, followed by details in section 4.3 and closing remarks in section 4.4.

4.1 Settings and Configurations

The specification of an experiment is done on two levels: *settings* and *configurations*. They serve two different purposes. Settings includes specifications that do not influence the results of the experiment directly. This includes configurations of the host machine such as enabling the GPU and more rarely changing settings such as where to store results.

Configurations, on the other hand, are directly related to the Python code. It is a JSON[1] format that fully specifies a model and the environment in which to run it. By "environment" we here mean an objective function whether this is a synthesized function allowing arbitrary draws or a data set consisting of a training and test set.

[1]JavaScript Object Notation: see `https://www.json.org/`.

4.1.1 The power of configurations

We are able to specify every configuration related to an experiment and subsequently run it *all within a Python notebook*. This is powerful because it circumvents changing a configuration file on the server for each experiment. Not only does this prevent mistakes in the setup but it also allows for much faster iterations.

By making specialized configurations easy we can leverage the cluster better. For instance, jobs that do not require a powerful CPU can specify the weaker requirement to exploit that the cluster probably has more of these instances available. Conversely, if we needed a large RAM requirement for the GPU this can be specified as exemplified in listing 4.1.

```
settings.WALLTIME = '00:40'
settings.QUEUE = 'gputitanxpascal'

run = execute(config_updates={
    'tag': 'heston',
    'obj_func': {
        'name': 'HestonOptionPricer',
    },
    'model': {
        'name': 'DKLGPModel',
        'kwargs': {
            'learning_rate': 0.1,
            'n_iter': 150,
        }
    },
    'gp_samples': 1000,
})
```

Listing 4.1 Example of specifying an experiment requiring GPU resources directly inside a Jupyter notebook.

4.2 Model execution

To run a model you have to specify a model and an environment in which to run it (that is e.g. a synthetic test function or a data set). This is all specified in a JSON format so it can be executed at another time and place: on a server or at a later point in time.

When an experiment runs it has three types of outputs: *metrics*, *artifacts* and *results*. Metrics is where streams of output is stored such as the training loss across

Table 4.1 Stored error metrics. See section 3.1.2 for a precise formulation.

MAE	Average L_1 error.
MAX	L_∞ error.
RMSE	Squared averaged L_2 error.
MNLP	Mean negative log probability.
NMSE	L_2 error normalized by the mean of the training outputs.

all iterations. Artifacts are file outputs we store as the run proceeds. This could be the posterior plot or the function plot in the feature space in the case of DKL (see fig. 3.8 for an example). Finally, results are computed and stored such as the final hyperparameters of the model and the errors on a test set. Table 4.1 includes an exhaustive list of the stored error metrics.

The reason for introducing abstract concepts such as metrics and artifacts becomes clear when we run it on a server as described in section 4.2.1. Then we are able to store these outputs on a hosted MongoDB database. This is made possible by the Python package, Sacred, which will also log the random seed and git version for reproducibility.

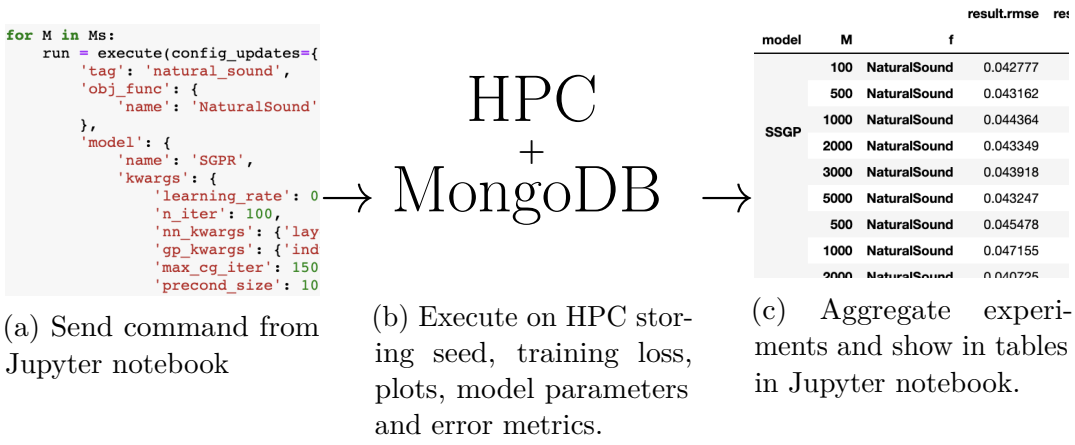
4.2.1 Server Execution

From within the same Jupyter notebook[2] we can configure a model, send it to a HPC for execution and retrieve the output in the same notebook for aggregation and inspection. The two notebook touch-points are tied together by a MongoDB that records the output of the server run. This workflow makes sure that everything associated with a collection of experiments can be collected in one place: this could be comparing the behavior of multiple models on the same test function. We illustrate the workflow in fig. 4.1.

Reproducability A seed is created and stored and the configuration is JSON formatted for reproducibility. The JSON format is also what enables running a notebook-specified experiment through the command-line—ultimately admitting it to be run on the server.

Aggregation Each run is associated with a model hash and an experiment hash uniquely determined by the model configuration and the (model, environment) configuration pair respectively. This way we can easily aggregate results.

[2]The Jupyter notebook is an interactive web-based environment to run and explore the output of our code: `https://jupyter-notebook.readthedocs.io`.



(a) Send command from Jupyter notebook

(b) Execute on HPC storing seed, training loss, plots, model parameters and error metrics.

(c) Aggregate experiments and show in tables in Jupyter notebook.

Fig. 4.1 Server execution flow.

Status Additionally the status such as FAILED and COMPLETED of a run can be directly inspected from the notebook along with the its current output and possible failure trace.

4.2.2 Local execution

Using exactly the same interface as for server execution we can specify an experiment to run locally. This can similarly be recorded in the MongoDB for aggregation. However, the main advantage is that it allows immediate access to, e.g., the model after a run for further inspection.

4.2.3 Execution Variants

The execution will automatically take care of a few tasks. For models which need to control the point selection (such as Adaptive Sparse Grid) sampling locations will be delegated to the model. For environments the data training data will be generated on the fly if it is a synthetic function. In the case of a `DataSet` it will use the specified split (see section 4.3.2 for these two types of environments).

4.3 Modules

Two modules of primary importance are *environments* and *models*. The former provides the environment from which the latter can draw both training and test observations. Both benefit from being composable–a concept we will elaborate promptly.

Table 4.2 Models. **Extra** refers to models not covered by the theory but for which there is still an implementation.

Category	Model	Description
Common	GPModel	GPy based.
	KISS-GP	GPyTorch based.
	DKL	GPyTorch based.
	SSGP	Sparse Spectrum Gaussian Process.
	SGPR	Variational inducing point GP.
	DNNBLR	-
	ActiveSubspace	-
	LASSO	-
	AdaptiveSparseGrid	Tasmanian based.
Extra	LocalLengthScaleGP	-
	LowRankGPModel	Low rank approximation written from scratch.
	QFF-GP	Quadrature Fourier Feature GP.
	RFF-GP	Sparse Spectrum Gaussian Process from scratch.
Helpers	SaveMixin	Interface for saving a model.
	TransformerModel	Used to combined GP and AS into AS-GP.
	NormalizerModel	Normalizes based on training input. Compatible with any model.

4.3.1 Models

For a complete list of `Models` at the time of writing see table 4.2. Essentially, a `Model` provides an interface for fitting a model and subsequently predicting with it.

Implementation-wise the helper functions are of particular interest. Using `TransformerModel` we can compose e.g. an Active Subspace and Gaussian Process model to obtain the AS-GP model. In a similar fashion `NormalizerModel` wraps any model and produces a version of the same interface that first normalizes the input.

The models implement the `SaveMixin` such that a complicated composed model such as `Normalizer<TransformerModel<ActiveSubspace, DKL>>` will correctly store (and retrieve) the normalized data, transformation matrix of the Active Subspace, PyTorch parameters of the DKL and the model structure.

4.3.2 Environments

For the experiments in chapter 3 we only used some of the implemented environments. Here we provide a high-level overview of the available varieties. For an exhaustive list we refer to the implementation.

- **Non-stationary** Sinusoidals with varying amplitude and length-scale.
- **Discontinous** Step functions and kinks.
- **Financial/Economic** simulated models such as the growth model and option pricing as well as a cleaned stock marked data sets.
- **UCI** Various (normalized) machine learning data ported from `https://people.orie.cornell.edu/andrew/code/`.
- **Natural Sound and Precipitation** Datasets of low dimensions with many observations ported from `https://github.com/kd383/GPML_SLD`.
- **Genz 1984** For integrands scalable to arbitrary dimensionality.
- **Optimization benchmarks** Synthetic functions such as Branin and Rosenbrock ported from GPyOpt.
- **Helpers** For automatically normalizing and creating embeddings.

In this context the helpers also demand further elaboration. Given any 1D environment we can construct an embedding which preserves the interface. Currently there is support for both linear embeddings $\mathbf{A}\boldsymbol{x}$ and embeddings of the form $\sum_i x_i^2$.

4.3.3 Configuration

A JSON configuration can be used to specify an experiment as mentioned in section 4.1. Internally each part of the configuration is handled by the associated module in a nested fashion. As an example, let us consider the following:

```
{
  'model': {
    'name': 'NormalizerModel',
    'kwargs': {
      'model': {
        'name': 'GPModel',
        'kwargs': {'learning_rate': 0.1},
      }
    }
  },
  ...
}
```

Listing 4.2 Example config

Here, the `NormalizerModel` class will be passed the associated `kwargs` through a call to the instance method `from_config` in order to construct an instance. It is the responsibility of that class to handle the construction of `model` which it does by finding the `GPModel` class and recursively call `from_config` with the associated `kwargs`.

There is always a corresponding programmatical way of specifying the same model:

```
NormalizerModel(model=GPModel(learning_rate=0.1))
```

Some of the parameters are lazy. In the following case this occurs because the input dimension, which the kernel depends on, is only known when the model is fitted:

```
SSGP(feature_extractor_constructor=LazyConstructor(RFFEmbedding, M=100))
```

As a configuration this is still specified in the same way as previously:

```
{
    'name': 'SSGP',
    'kwargs': {
      'feature_extractor_constructor': {
        'name': 'RFFEmbedding',
        'kwargs': {'M': 100},
      }
    }
}
```

This very modular approach to configuration ensures that the configuration stays very close to the implementation and allows additional parameters to be added with minimal overhead when experimenting.

4.3.4 Context

One might wonder what handles the top level parsing of the example configuration in listing 4.2. This is the top level `Context` module which is a minimal sub-class of the `ConfigMixin`. It provides a centralized place for a given execution to access all *initialised* instances imposed by a configuration.

4.4 Conclusing remarks

We have settled on an implementation that is very useful for the specific type of experiments we are interest in carrying out. However, most of problem-specific code is

associated with the implementation of particular environments and models.[3] We have tried to keep the *structure*, such as environments and models, rather generic. A lot of this generalization occurred along the way as new requirements came along. We recognize that we need to be exposed to even more cases but our hope is to eventually extract these components into their own library.

[3]Note that it is also still very tied to the particular choice of the HPC environment. In that sense it should still be considered a prototype.

Chapter 5

Discussion

This chapter will discuss the main results and suggest possible future directions.

5.1 High-dimensionality

We primarily see three directions for dealing with high-dimensionality: extending DKL, investigating other model assumptions or finding more theoretically founded models. We will now cover these in order.

We are restricted to fixing the dimensionality of the feature space for Deep Kernel Learning ex-ante. However, we did observe that, even in a high dimensional feature space, points would be consolidated on a line segment if the intrinsic dimensionality was indeed 1. Regardless, further reduction was not trivial since the line segment was not for example necessarily axis-aligned. Ideally we want to learn the effective dimensionality– especially when coupled with scalable methods since the number of inducing points needed for KISS-GP grows exponentially with dimensionality. One possible direction to explore is to determine the effective dimension through Bayesian mixture modeling [Reich et al.]. One should be careful, though, as it comes at a heavy computational cost.

We managed to make the training procedure of DKL robust by several tricks. However, this came at a high cost of complexity of the procedure. Understanding how to make the training loss more stable would not only make us more confident in the model's robustness but will most likely also lead to better generalization.

DKL was able to learn a linear embedding in the value iteration (section 3.5.2) and more general embeddings (section 3.3.1). The manifold assumption made in DKL may be too strong for some real data sets. Instead we might be able to exploit some other structure such as additive decomposition of the response $f(x) = \sum_{j=1}^{G} f^{(j)}\left(x^{(j)}\right)$ where

each $x^{(j)}$ lives in a low-dimensional subspace [Duvenaud et al.]. Promising results for the usefulness of additive GPs have been shown empirically in the Bayesian optimization setting even when the modelling assumption does not hold [Kandasamy et al.]. Whether these observations translate into pure regression problems is not entirely certain, though, since they are most likely only beneficial in regimes with few samples.

Even though DKL has been shown to work well in practice, it relies on neural networks for which very few guarantees are available. Further, DKL required substantial engineering tricks to work. Therefore, if the objective function is *known* to be a linear embedding and of a form that requires less than $10,000$ observations, then AS provides a much more robust approach. If this is not the case, exploring more principled ways could be of interest. It is natural to expand on the well-founded idea of Active Subspaces. In [Bilionis et al.] they do just that by absorbing the AS projection matrix $\mathbf{W}$ into the hyperparameters of a GP and further make the method gradient free. These methods are still restricted to linear manifolds, however. Very recently [Bridges et al.] introduced the theoretically justified Active Manifold method which relies on an iterative local analysis rather than global summary statistics like Active Subspaces. Combining this with GP for joint learning could be a promising direction.

5.2 Scalability

If the data set is medium sized ($[10^4, 10^5]$) then we advise using inducing point methods such as SGPR and SSGP. Only when the scalability of these methods breaks down when considering even larger data sets would we suggest using KISS-GP. This is of course assuming all data points are relevant. Assessing how informative each data point is as we increase the sizes of the training data sets is not easy and heavily influences what approximate model to use. That is, if a few points summarize the data set well, using a few *carefully selected* inducing points as with SGPR would suffice.

However if we were to use SSGP we note that it suffers from variance starvation (fig. 3.10). This is a consequence of fixing the spectral points—no matter whether this is done through random sampling or optimization. Instead we could approximate the integral using numerical integration techniques such as Gauss-Hermite as in [Dao et al.] and more recently in the Bayesian optimization setting [Mutny and Krause].[1] It would be interesting to study the applicability of SSGP with this correction.

Reliable estimates are especially important when injecting the model into an iterative value iteration scheme in which an error can propagate. Very recently exact

[1] Our code base already includes an implementation of this for the SE kernel.

GP regression was carried out for massive data sets due to advances in conjugate gradient methods [43]. They circumvent the memory bottleneck we faced in section 3.5.1 by considering batches of training data. These results suggest that for a fixed budget it might be beneficial to spend computational resources on an exact GP rather than increasing the number of inducing points. Understanding this tension would be useful.

5.3 Uncertainty Quantification

By considering DKL and DNNBLR we have effectively investigated neural networks in which we only place a prior distribution on the weights of the last layer (specifically a gaussian). It is natural to consider modelling *all* weights with a distribution. This is well studied in the literature under the name *Bayesian neural network* (BNN). However, they are notoriously difficult for doing inference because of their high-dimensional prior with possibly intricate covariance structure.[2] Still, BNNs might prove to provide better confidence bounds by shying away from the point estimates otherwise made in neural network counterparts by their weight initialization.

5.4 Economic settings

For problems that are not statistically challenging vanilla GPs still provide a very robust approach to modelling both the mean and confidence interval reliably. Even when discontinuities were present, as in the case of the policy functions in section 3.5.3, it still performed competitively with respect to the L_∞. Conversely DKL was not as robust both across the problems but also with respect to the random initialization. Further, it required a significant amount of fine-tuning to work. We will therefore make the case for GPs when we do not suffer from the curse of dimensionality in its various varieties (i.e. low length-scale, high noise, or high dimensionality).

For the growth model we observed how we could asymptotically more cheaply learn the low dimensional manifold. For this purpose DKL was promising for scaling to new problem sizes.

We have to be very careful when comparing model performance. In comparing KISS-GP with exact GP for Option Pricing in section 3.5.1 we initially set the learning rate and epochs too low for the Adam optimizer to converge. By the optimization effectively becoming a bottleneck for performance both models would perform equally well. We mention in passing that it indeed turned out that KISS-GP was competitive

[2]For a thorough overview see [Gal].

but this is not the point. We imagine this could be a problem in the literature as well where the convergence of the commonly used BFGS is usually not examined.

5.5 Conclusion

We set out to provide well-calibrated uncertainty estimates in economic applications. To this end, it was natural to consider GPs which raised three challenges when facing the requirements of the economic setting: non-stationarity, high-dimensionality and scalability in observations.

To handle non-stationarity and high-dimensionality we studied Deep Kernel Learning, which we compared with both GPs and neural network counterparts with and without a Bayesian linear regressor. We further compare with the existing state-of-the-art model in the high-dimensional economic settings: Active Subspaces with Gaussian Processes. We showed empirically that it is only interesting to consider DKL when either the assumptions made by this model breaks down or the computational cost becomes prohibitively large.

For tackling scalability we focused on inducing point methods and KISS-GP. Combined with DKL it is possible to apply KISS-GP even to high-dimensional problems. As a proof-of-concept we applied this approach to a stochastic optimal growth model suggesting that the method could scale to unprecedented dimensionality greater than 500.

In this thesis we have laid the groundwork for applying these expressive GP models in an economic setting. This includes a code base for running the various models in parallel and a detailed description of them. We believe this will prove valuable as a basis for further studies.

References

[Bellman] Bellman, R. A Markovian decision process. pages 679–684.

[Bengio et al.] Bengio, Y., Delalleau, O., and Roux, N. L. The Curse of Highly Variable Functions for Local Kernel Machines. In Weiss, Y., Schölkopf, B., and Platt, J. C., editors, *Advances in Neural Information Processing Systems 18*, pages 107–114. MIT Press.

[Bilionis et al.] Bilionis, I., Tripathy, R., and Gonzalez, M. Gaussian processes with built-in dimensionality reduction: Applications in high-dimensional uncertainty propagation.

[Bridges et al.] Bridges, R. A., Gruber, A. D., Felder, C., Verma, M., and Hoff, C. Active Manifolds: A non-linear analogue to Active Subspaces.

[Brumm and Scheidegger] Brumm, J. and Scheidegger, S. Using Adaptive Sparse Grids to Solve High-Dimensional Dynamic Models. 85(5):1575–1612.

[Cai and Judd] Cai, Y. and Judd, K. L. Advances in numerical dynamic programming and new applications. In *Handbook of Computational Economics*, volume 3, pages 479–516. Elsevier.

[Calandra et al.] Calandra, R., Peters, J., Rasmussen, C. E., and Deisenroth, M. P. Manifold Gaussian Processes for regression. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 3338–3345. IEEE.

[Chan and Ng] Chan, R. H. and Ng, M. K. Conjugate gradient methods for Toeplitz systems. 38(3):427–482.

[Constantine] Constantine, P. G. *Active Subspaces: Emerging Ideas for Dimension Reduction in Parameter Studies*, volume 2. SIAM.

[Cunningham et al.] Cunningham, J. P., Shenoy, K. V., and Sahani, M. Fast Gaussian process methods for point process intensity estimation. In *Proceedings of the 25th International Conference on Machine Learning - ICML '08*, pages 192–199. ACM Press.

[Cutajar et al.] Cutajar, K., Osborne, M. A., Cunningham, J. P., and Filippone, M. Preconditioning Kernel Matrices.

[Dao et al.] Dao, T., De Sa, C. M., and Ré, C. Gaussian Quadrature for Kernel Features. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6107–6117. Curran Associates, Inc.

[Duvenaud et al.] Duvenaud, D., Nickisch, H., and Rasmussen, C. E. Additive Gaussian Processes. page 9.

[Duvenaud] Duvenaud, D. K. Automatic Model Construction with Gaussian Processes. page 157.

[Fritz et al.] Fritz, J., Neuweiler, I., and Nowak, W. Application of FFT-based algorithms for large-scale universal kriging problems. 41(5):509–533.

[Gal] Gal, Y. Uncertainty in Deep Learning. page 174.

[Gardner et al.] Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., and Wilson, A. G. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration.

[Gelman et al.] Gelman, A., Hwang, J., and Vehtari, A. Understanding predictive information criteria for Bayesian models.

[Hornik et al.] Hornik, K., Stinchcombe, M., and White, H. Multilayer Feedforward Networks Are Universal Approximators. 2(5):359–366.

[Juillard and Villemot] Juillard, M. and Villemot, S. Multi-country real business cycle models: Accuracy tests and test bench. 35(2):178–185.

[Kandasamy et al.] Kandasamy, K., Schneider, J., and Poczos, B. High Dimensional Bayesian Optimisation and Bandits via Additive Models.

[Kingma and Ba] Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization.

[Kubler and Scheidegger] Kubler, F. and Scheidegger, S. Self-justified equilibria: Existence and computation. page 40.

[Maas et al.] Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier Nonlinearities Improve Neural Network Acoustic Models. page 6.

[McIntire et al.] McIntire, M., Ratner, D., and Ermon, S. Sparse Gaussian Processes for Bayesian Optimization. In *UAI*.

[Mutny and Krause] Mutny, M. and Krause, A. Efficient High Dimensional Bayesian Optimization with Additivity and Quadrature Fourier Features. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 9005–9016. Curran Associates, Inc.

[Neal] Neal, R. M. *Bayesian Learning for Neural Networks*, volume 118 of *Lecture Notes in Statistics*. Springer New York.

[Osborne et al.] Osborne, M., Garnett, R., Ghahramani, Z., Duvenaud, D. K., Roberts, S. J., and Rasmussen, C. E. Active Learning of Model Evidence Using Bayesian Quadrature. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 46–54. Curran Associates, Inc.

[Pleiss et al.] Pleiss, G., Gardner, J. R., Weinberger, K. Q., and Wilson, A. G. Constant-Time Predictive Distributions for Gaussian Processes.

[Quiñonero-Candela and Rasmussen] Quiñonero-Candela, J. and Rasmussen, C. E. A unifying view of sparse approximate Gaussian process regression. 6:1939–1959.

[Rahimi and Recht] Rahimi, A. and Recht, B. Random Features for Large-Scale Kernel Machines. page 8.

[Rasmussen and Williams] Rasmussen, C. E. and Williams, C. K. I. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press. OCLC: ocm61285753.

[Reich et al.] Reich, B. J., Bondell, H. D., and Li, L. Sufficient Dimension Reduction via Bayesian Mixture Modeling. 67(3):886–895.

[Rudin] Rudin, W. *Fourier Analysis on Groups*, volume 121967. Wiley Online Library.

[Saatci] Saatci, Y. Scalable Inference for Structured Gaussian Process Models. page 184.

[Scheidegger and Bilionis] Scheidegger, S. and Bilionis, I. Machine learning for high-dimensional dynamic stochastic economies. page 46.

[Shen et al.] Shen, Y., Seeger, M., and Ng, A. Y. Fast Gaussian Process Regression using KD-Trees. In Weiss, Y., Schölkopf, B., and Platt, J. C., editors, *Advances in Neural Information Processing Systems 18*, pages 1225–1232. MIT Press.

[Smith] Smith, R. C. *Uncertainty Quantification: Theory, Implementation, and Applications*, volume 12. Siam.

[Snelson et al.] Snelson, E., Rasmussen, C. E., and Ghahramani, Z. Warped Gaussian Processes. page 8.

[Snoek et al.] Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M. M. A., Prabhat, and Adams, R. P. Scalable Bayesian Optimization Using Deep Neural Networks.

[Titsias] Titsias, M. K. Variational Learning of Inducing Variables in Sparse Gaussian Processes. page 8.

[Turner] Turner, R. E. Statistical models for natural sounds.

[43] Wang, K. A., Pleiss, G., Gardner, J. R., Tyree, S., Weinberger, K. Q., and Wilson, A. G. Exact Gaussian Processes on a Million Data Points.

[44] Wang, Z., Gehring, C., Kohli, P., and Jegelka, S. Batched large-scale bayesian optimization in high-dimensional spaces.

[45] Wilson, A. G., Hu, Z., Salakhutdinov, R., and Xing, E. P. Deep Kernel Learning.

[46] Wilson, A. G., Nehorai, A., Gilboa, E., and Cunningham, J. P. Fast Kernel Learning for Multidimensional Pattern Extrapolation. page 18.

[Wilson and Nickisch] Wilson, A. G. and Nickisch, H. Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP). page 10.

Appendix A

Theory

A.1 Linear Algebra

A.1.1 Woodbury, Sherman and Morrison Formula

The matrix inversion lemma states that

$$\left(\mathbf{A}+\mathbf{UBV}^{\top}\right)^{-1}=\mathbf{A}^{-1}-\mathbf{A}^{-1}\mathbf{U}\left(\mathbf{B}^{-1}+\mathbf{V}^{\top}\mathbf{A}^{-1}\mathbf{U}\right)^{-1}\mathbf{V}^{\top}\mathbf{A}^{-1}, \tag{A.1}$$

where $\mathbf{A}$ and $\mathbf{B}$ are square matrices of possibly different dimensions. If this is the case the equality provides a way to replace the inversion of a big matrix with a much smaller one. It follows directly from application of the blockwise matrix inversion in section A.1.3.

A.1.2 Sylvester's Determinant Identity

A analog to the matrix inversion lemma for determinants exists that will similarly permit computational speedup

$$\left|\mathbf{A}+\mathbf{UBV}^{\top}\right|=|\mathbf{A}||\mathbf{B}|\left|\mathbf{B}^{-1}+\mathbf{V}^{\top}\mathbf{A}^{-1}\mathbf{U}\right|. \tag{A.2}$$

A.1.3 Inverse of a partitioned matrix

Let $\mathbf{A}$ and its inverse $\mathbf{A}^{-1}$ be written as a block partition

$$\mathbf{A}=\begin{bmatrix}\mathbf{F} & \mathbf{G}\\ \mathbf{G}^{\top} & \mathbf{H}\end{bmatrix},\quad \mathbf{A}^{-1}=\begin{bmatrix}\tilde{\mathbf{F}} & \tilde{\mathbf{G}}\\ \tilde{\mathbf{G}}^{\top} & \tilde{\mathbf{H}}\end{bmatrix} \tag{A.3}$$

where $\mathbf{F}$ and $\tilde{\mathbf{F}}$ are square matrices of same dimensionality $n_1 \times n_1$ and $\mathbf{H}$ and $\tilde{\mathbf{H}}$ are similarly $n_2 \times n_2$ matrices. Then the submatrices of the inverse are given by

$$\begin{aligned} \tilde{\mathbf{F}} &= \mathbf{F}^{-1} + \mathbf{F}^{-1}\mathbf{G}\mathbf{M}\mathbf{G}^\top\mathbf{F}^{-1} \\ \tilde{\mathbf{G}} &= -\mathbf{F}^{-1}\mathbf{G}\mathbf{M} \\ \tilde{\mathbf{H}} &= \mathbf{M} \end{aligned} \tag{A.4}$$

with $\mathbf{M} = \left(\mathbf{H} - \mathbf{G}^\top\mathbf{F}^{-1}\mathbf{G}\right)^{-1}$.

A.1.4 Marginalization of product

Suppose $\boldsymbol{f}$ is zero mean gaussian distributed

$$p(\mathbf{f}) = \mathcal{N}(\mathbf{0}, \mathbf{B}) \tag{A.5}$$

and another variable $\boldsymbol{y}$ is multivariate gaussian distributed as well but with a mean that is a linear transformation of $\boldsymbol{f}$

$$p(\mathbf{y}|\mathbf{f}) = \mathcal{N}(\mathbf{V}\mathbf{f}, \mathbf{A}). \tag{A.6}$$

Then the marginal distribution of $\boldsymbol{y}$ is given as

$$\begin{aligned} p(\mathbf{y}) &= \int \mathrm{d}\mathbf{f} p(\mathbf{y}|\mathbf{f}) p(\mathbf{f}) \\ &= \mathcal{N}\left(\mathbf{0}, \mathbf{A} + \mathbf{V}\mathbf{B}\mathbf{V}^\top\right). \end{aligned} \tag{A.7}$$

A.1.5 Norms

Operator norm:

$$\|A\|_p = \sup_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}. \tag{A.8}$$

Appendix B

Practical Aspects

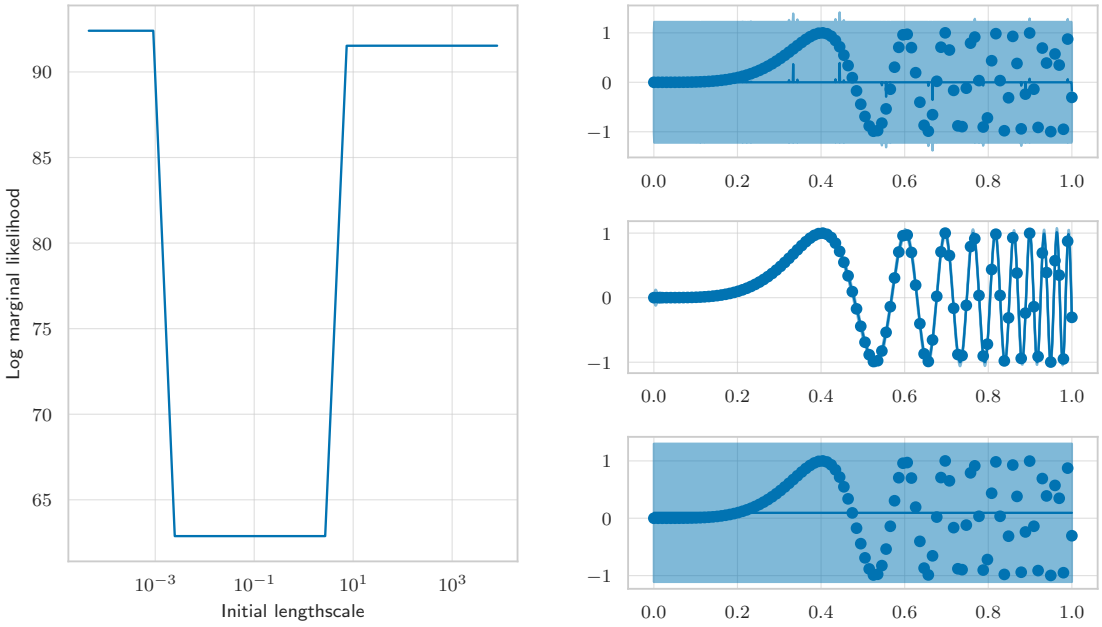
The theory leaves out several important aspects when applying GPs in practice. This section provides guides on more practical aspects such as normalization, initialization and numerical instability.

B.1 Initialization

Reasonable initialization of the hyperparameters is require to ensure convergence. Figure B.1 illustrates how extreme initializations can lead to the length-scale tending either to too small values or to values larger than the domain interval. This motivates normalization (covered in Appendix B.2 and Appendix B.3) as it would otherwise not be possible to guess reasonable hyperparameters across functions without inspecting them first.

B.2 Normalization

For clarity, usual data transformations such as normalization have been left out of the presentation. It is however important to point out their importance. Without data augmentation many fixed hyperparameters of our models would not work across different problem domains. Take for example the step size of the Adam optimizer used for DKL from section 2.4.1. The convergence would suffer if a correctly scaled domain was scaled up since the steps would be relatively smaller. We could choose to have these hyperparameters depend on the input. However, as there are many such variables with the same type of dependency it is much easier to simply rescale the input and



(a) (b)

Fig. B.1 A GP can get stuck in a local optimum given too extreme initial length-scales. a) The effect of the length-scale on the marginal likelihood. b) The posterior for the three different regions where the uppermost figure is associated with left-most region in (a) and so forth.

output space. One such example is the zero mean prior we assumed in our GP which is now much more reasonable.

To be precise denote the standardizations as

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \hat{\mu}_X}{\hat{\sigma}_X} \quad \text{and} \quad \tilde{\mathbf{Y}} = \frac{\mathbf{Y} - \hat{\mu}_Y}{\hat{\sigma}_Y}, \tag{B.1}$$

where $\hat{\mu}$ and $\hat{\sigma}$ are the empirical mean and standard deviation respectively.

We then train the model on $\tilde{\mathbf{X}}$ and $\tilde{\mathbf{Y}}$ instead. Subsequent points can then be transformed using the associated $\hat{\mu}$ and $\hat{\sigma}$ to each data matrix. Further, we later recover the predictions $\tilde{\mathbf{P}}$ made in the transformed space by denormalizing them with

$$\mathbf{P} = \tilde{\mathbf{P}}\hat{\sigma}_Y + \hat{\mu}_Y. \tag{B.2}$$

B.3 Normalization of the feature space

In Appendix B.2 we argue for the importance of normalizing the input space for the GP. This argument carries over to normalization of the feature space as well when considering DKL. This is because the kernel can be viewed as an input transformation on which a regular stationary kernel is fitted. So if we do not scale the feature space, even if we were to scale the input space, it would be impossible to set reasonable priors over hyperparameters (the importance of which is described in Appendix B.1).

Conceptually this rescaling can be absorbed into the network and treated as a network with the additional constraints on the output.[1]

The importance of the rescaling can be observed experimentally. DKL, for which Conjugate Gradient method is used, requires too many iterations to converge. This leads to wildly overestimated posterior variance and fluctuating mean. An example of such an extreme failure case can be found in fig. B.2a.

B.4 Hyperparameter Optimization

When using Adam to optimize both neural network parameters and hyperparameters of a probabilistic model, as in DKL and DNNBLR, we observed an increasingly unstable

[1] In practice the normalization of the network output is done using a BatchNorm layer. When the batch is the entire training set (as it is in our case) the moving average it keeps over training batches corresponds to calculating a mean and standard deviation over the entire training set. During predictions these statistics are then used to normalize the input. Only a minor impact on running time was observed so correctness of the implementation was prioritized.

loss. That is, the loss plot had spikes which were increasing in both frequency and amplitude over time. Several method were employed to prevent this, such as gradient clipping and adjusting the learning rate. However, this did not completely mitigate the spikes.

To prevent termination with a high loss we modify the assumption of having fixed number of epochs, T. That is, instead of yielding the last parameter-configuration $\boldsymbol{\theta}_T$ of the network we use the network configuration $\boldsymbol{\theta}_t$ where $t < T$ is the iteration *with the lowest loss.*

B.5 Loose Stopping Critierion during Training

One computational benefit of using the Conjugate Gradient method (section 2.2.2.6) is that we have a criteria for stopping the iterative scheme early on. That is, if the relative residual

$$\left\|(\mathbf{K} - \sigma^2\mathbf{I})\mathbf{v}^* - \mathbf{y}\right\| / \|\mathbf{y}\|$$

is low it suggests that the Conjugate Gradient iterations have converged. So we choose some ϵ as a threshold for when to stop the iterations.

To calculate predictions choosing a small enough $\epsilon \leq 0.01$ is critical for getting sufficient accuracy. However, when we learn the hyperparameters, the accuracy of the log marginal likelihood is only important to the extend that it allows us to find a good maximum. If we learn the hyperparameters with a stochastic optimizer like Adam [Kingma and Ba] which is robust against noise in the gradients then a much looser convergence criterion of $\epsilon \approx 1$ was still observed to find a good hyperparameter configuration [43].

Recommendation Choose high CG stopping criterion during training, $\epsilon = 1$.

B.5.0.1 Double Precision

In most cases we observed that double precision when using Conjugate Gradients (section 2.2.2.6) is dispensable—especially when considering that the same calculation done with floating points allows for more iterations. However, this is only the case if the learned hyperparameters can indeed be expressed in low precision. For extremely low noise settings ($\sigma^2_{\text{noise}} \leq 10^{-4}$) floating points would prevent us from learning the true noise-level, effectively providing a worse model.

Recommendation Use double precision only if the true hyperparameters is assumed to require the precision.

B.6 Trouble-shooting Low Noise

In the low noise setting both the Conjugate Gradient and Cholesky approach will break down without double precision (this includes the noise-free setting where a lower bound $\sigma^2_{\text{noise}} = 10^{-4}$ was used in practice). The increased precision results in a noticeable slow-down, however. Alternatively one could add noise to the observations, but in our experience this required adding noise of order 10^{-2} when the output space is $[0, 1]$. This had substantial impact on the performance for tricky high-oscillation functions. Instead we consider a few approaches possible when using CG, where the problem shows up as a convergence issue because of an ill-conditioned matrix.

The precision problem can be divided into two categories: 1) failure during training when CG does not converge when calculating the marginal likelihood and 2) successful training but CG fails to converge for prediction. The latter was by far the most commonly observed problem and turns out to have an interesting solution. Note that these two problems also differ by the first being a hard failure in which the posterior is uninformative, while the latter simply leads to a prediction that is underconfident and less smooth then what is expected with our prior (see fig. B.2 for a concrete example of this difference).

For failures only at prediction time it turns out that approximating the posterior variance with Lanczos Variance Estimates [Pleiss et al.] will yield a more accurate variance. In other words, this approximation will actually be closer to the double precision prediction then a model also using low precision but without the variance estimate. This may sound somewhat surprising considering it is an approximation but note that in practice numerical stability matters as well as the theoretical approximation. In our case we are trading off the distance from our true kernel to gain numerical stability.

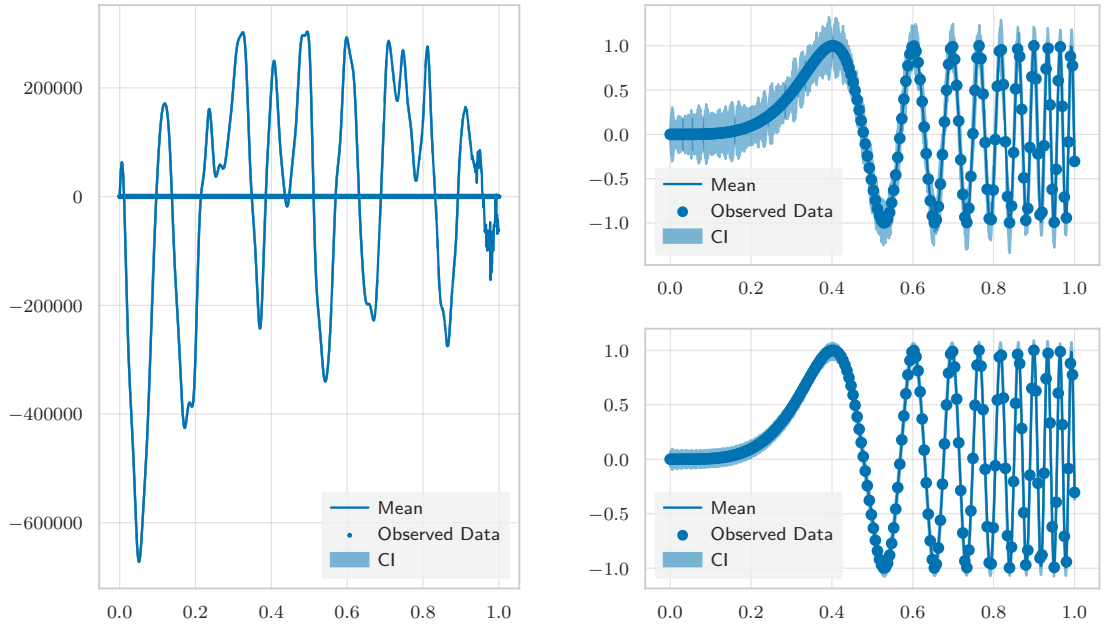
To understand why Lanczos Variance Estimates might improve the posterior first note that the numerical instability is caused by the kernel $\mathbf{K} + \sigma^2_{\text{noise}}\mathbf{I}$ having extremely small eigenvalues in the low noise setting. As σ^2_{noise} approaches zero we are practically finding the eigenvalues of $\mathbf{K}$. Especially if we have many observations some of these observations will be redundant—or equivalently some eigenvalues will be extremely small. We believe Lanczos Variance Estimates mitigates this since it stops when a sufficiently accurate low rank approximation of $\mathbf{K} + \sigma^2_{\text{noise}}\mathbf{I}$ have been found. This

effectively cuts off the smallest eigenvalues that led to the ill-conditioning. For further details on the method see [Pleiss et al.].

In many cases using Lanczos Variance Estimates has proven sufficient in fixing the issue with low noise. However, for the cases that fail during training we consider some practical advice for tuning the CG hyperparameters to help with convergence. Of the varies hyperparameters increasing the size of preconditioner for CG will have the biggest impact. In fact, the convergence rate of CG will improve exponentially with the size of the preconditioner for 1D SE kernels [Gardner et al., Theorem 1]. However, we observed it had its own numerical issues for sizes above 80. To this end, we also consider simply increasing the number of CG steps. Figure B.2a illustrates an extreme case of devoting too little computational effort to the CG procedure.

Finally it should be noted that using a properly tuned CG might have its benefits over enabling double precision and using Cholesky. It has been observed empirically that CG can sometimes provide better residuals than Cholesky using 32-bit floating points [Gardner et al., Figure 1].

Recommendation For our problems we experienced that a preconditioner size of 20 and a maximum step size of 10,000 was sufficient. Note that the maximum steps were rarely reached as a threshold from Appendix B.5 ensures early stopping.



(a) Failure to converge during training.

(b) The numerical instability during testing (top) is mitigated by using Lanczos Variance Estimates (bottom).

Fig. B.2 Numerical instability when using CG in a low-noise setting.

Appendix C

Experiments

C.0.1 Embeddings

The results for embeddings covered in section 3.3.1 are displayed in table C.1.

C.0.2 High-dimensional Policy Function

MNLP for High-dimensional Policy Function section 3.5.3 can be found in table C.2.

C.0.3 Kernel reconstruction

Our error on the kernel is not uniform (otherwise we could easily correct it). This section shows the effect on the covariance matrix given our different approximations. To this end, we generate a RBF covariance matrix from a 1000 points sampled from $\mathcal{N}(0, 1)$.

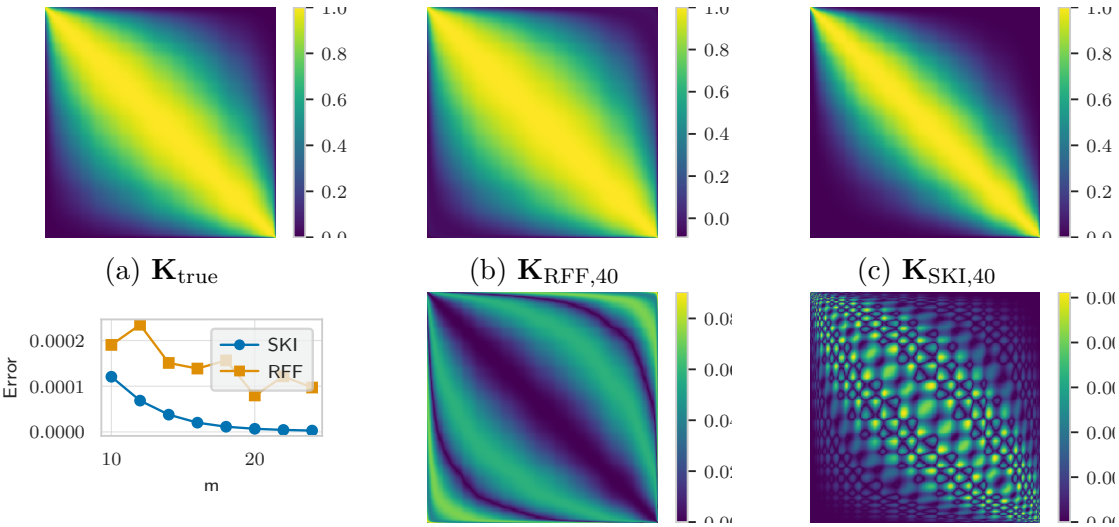
Model	D	RMSE	L_∞	MNLP
AS-GP	**50**	0.00150 ± 0.00017	0.007968 ± 0.000772	-2.341 ± 0.062
	100	0.00070 ± 0.00024	0.003307 ± 0.001315	-2.763 ± 0.171
	500	0.00015 ± 0.00004	0.000711 ± 0.000150	-3.508 ± 0.137
	1000	0.00008 ± 0.00001	0.000360 ± 0.000058	-3.882 ± 0.109
DKL [1000-500-50-1]	**50**	0.00184 ± 0.00006	0.038482 ± 0.009060	-1.796 ± 0.045
	100	0.00226 ± 0.00106	0.025174 ± 0.014882	-1.805 ± 0.130
	500	0.00253 ± 0.00167	0.016710 ± 0.016364	-1.731 ± 0.067
	1000	0.00478 ± 0.00308	0.029156 ± 0.022973	-1.632 ± 0.118
DKL [1]	**50**	0.00042 ± 0.00002	0.002468 ± 0.000952	-2.947 ± 0.016
	100	0.00038 ± 0.00004	0.002101 ± 0.000468	-3.003 ± 0.032
	500	0.00049 ± 0.00009	0.002372 ± 0.000512	-2.941 ± 0.089
	1000	0.00055 ± 0.00003	0.002632 ± 0.000132	-2.893 ± 0.029
DNN [1000-500-50-1]	**50**	0.03402 ± 0.02328	0.182243 ± 0.057918	nan ± 0.000
	100	0.02786 ± 0.00688	0.126487 ± 0.013855	nan ± 0.000
	500	0.03712 ± 0.00327	0.126506 ± 0.012235	nan ± 0.000
	1000	0.07013 ± 0.06383	0.217671 ± 0.156333	nan ± 0.000
DNNBLR [1000-500-50-1]	**50**	0.00526 ± 0.00143	0.061852 ± 0.008545	-1.684 ± 0.076
	100	0.00902 ± 0.00672	0.060456 ± 0.029185	-1.241 ± 0.823
	500	0.00751 ± 0.00628	0.050739 ± 0.051432	-1.432 ± 0.567
	1000	0.00414 ± 0.00120	0.020397 ± 0.006771	-1.711 ± 0.050
LASSO	**50**	0.43511 ± 0.00020	1.332370 ± 0.012745	nan ± 0.000
	100	0.43486 ± 0.00031	1.141679 ± 0.013556	nan ± 0.000
	500	0.43254 ± 0.00004	1.019264 ± 0.004758	nan ± 0.000
	1000	0.43120 ± 0.00003	0.978708 ± 0.002359	nan ± 0.000

Model	D	RMSE	L_∞	MNLP
AS-GP	**50**	0.00159 ± 0.00025	0.013049 ± 0.003393	-1.827 ± 0.009
	100	0.00088 ± 0.00030	0.005433 ± 0.002367	-1.838 ± 0.005
	500	0.00027 ± 0.00007	0.001828 ± 0.000211	-1.844 ± 0.001
	1000	0.00024 ± 0.00006	0.001241 ± 0.000507	-1.843 ± 0.003
DKL [1000-500-50-1]	**50**	0.01423 ± 0.00037	0.090630 ± 0.007809	-0.999 ± 0.064
	100	0.01877 ± 0.00604	0.092191 ± 0.034484	-0.190 ± 1.041
	500	0.01206 ± 0.00381	0.062293 ± 0.020283	-1.251 ± 0.304
	1000	0.00966 ± 0.00103	0.044322 ± 0.013840	-1.429 ± 0.057
DKL [1]	**50**	0.00267 ± 0.00021	0.015169 ± 0.006114	-1.820 ± 0.013
	100	0.00263 ± 0.00021	0.012821 ± 0.001388	-1.815 ± 0.010
	500	0.00272 ± 0.00004	0.013331 ± 0.000749	-1.814 ± 0.002
	1000	0.00273 ± 0.00003	0.014878 ± 0.002171	-1.812 ± 0.001
DNN [1000-500-50-1]	**50**	0.03005 ± 0.00295	0.169829 ± 0.022830	nan ± 0.000
	100	0.02570 ± 0.01063	0.107483 ± 0.025201	nan ± 0.000
	500	0.32473 ± 0.51634	1.503797 ± 2.390889	nan ± 0.000
	1000	0.04113 ± 0.00472	0.141705 ± 0.018248	nan ± 0.000
DNNBLR [1000-500-50-1]	**50**	0.01869 ± 0.00197	0.092794 ± 0.005984	-0.060 ± 0.455
	100	0.01697 ± 0.00093	0.082912 ± 0.021703	-0.579 ± 0.147
	500	0.01177 ± 0.00037	0.055253 ± 0.008278	-1.224 ± 0.038
	1000	0.01538 ± 0.00673	0.084948 ± 0.061728	-0.840 ± 0.794
LASSO	**50**	0.43496 ± 0.00004	1.340434 ± 0.005684	nan ± 0.000
	100	0.43475 ± 0.00012	1.136961 ± 0.013889	nan ± 0.000
	500	0.43262 ± 0.00006	1.013679 ± 0.002826	nan ± 0.000
	1000	0.43124 ± 0.00004	0.976224 ± 0.002507	nan ± 0.000

Table C.1 Results for embedding illustrated in fig. 3.4. *Top* shows the noise free setting while the *bottom* table shows results from a noisy setting with $\epsilon = 0.01$. Note that AS-GP is still provided noise-free gradients in the latter setting. All experiments shows mean ± one standard deviation computed over 5 runs.

Table C.2 MNLP for 2-20 dimensional discontinuous policy functions. DKL and DNNBLR both uses a [1000-500-50-2] network. DNN and LASSO are not included since they do not provide sufficient statistics. All experiments shows mean ± one standard deviation computed over 5 runs.

	MNLP				
Dimensionality	2	4	8	12	20
Model					
GP (Matern)	-4.41 ± 0.01	-2.57 ± 0.01	-2.59 ± 0.00	-2.50 ± 0.02	-2.67 ± 0.02
GP (SE)	990.36 ± 509.06	-2.06 ± 0.03	-2.16 ± 0.20	-2.38 ± 0.04	-2.34 ± 0.05
DKL (Matern)	-4.31 ± 0.03	0.10 ± 0.70	1.33 ± 0.41	0.28 ± 0.29	-0.50 ± 0.31
DKL (SE)	-4.22 ± 0.08	0.20 ± 0.49	0.90 ± 0.21	-0.00 ± 0.30	0.05 ± 0.46
DNNBLR	-0.26 ± 2.71	-0.26 ± 0.48	-1.19 ± 0.42	-1.72 ± 0.31	-0.48 ± 0.48



(a) $\mathbf{K}_{\text{true}}$ (b) $\mathbf{K}_{\text{RFF},40}$ (c) $\mathbf{K}_{\text{SKI},40}$

(d) $||\mathbf{K}_{\text{true}} - \mathbf{K}||_2$ vs m. (e) $|\mathbf{K}_{\text{true}} - \mathbf{K}_{\text{RFF}}|$ (f) $|\mathbf{K}_{\text{true}} - \mathbf{K}_{\text{SKI}}|$

Fig. C.1 a) shows a RBF kernel on 1000 sorted points generated from $\mathcal{N}(0, 1)$. Subsequent matrices shows the approximation and associated error for RFF and SKI using 40 inducing points.