Imperial College London
Department of Computing

Hierarchical Gaussian Processes for Large Scale Bayesian Regression

Jun Wei Ng

Supervisor: Dr. Marc P. Deisenroth

Second Marker: Dr. A. Aldo Faisal

Submitted in partial fulfilment of the requirements for the
MSc Degree in Computing Science of Imperial College London

September 2014

Abstract

We present a deep hierarchical mixture-of-experts model for scalable Gaussian process (GP) regression, which allows for highly parallel and distributed computation on a large number of computational units, enabling the application of GP modelling to large data sets with tens of millions of data points without an explicit sparse representation. The key to the model is the subdivision of the data at each node into multiple child GPs performing independent computations, each with a subset of the training data. All local models share the same set of hyperparameters are trained jointly. We assume independence amongst child GPs, but this approximation is mitigated by having child GPs with overlapping subsets to account for the covariance of data in different subsets. We show that our method can match the performance of a full GP (in both prediction and hyperparameter selection by marginal likelihood maximisation) with a significantly lower computational and memory cost. We outline a number of methods for the subdivision of GPs at each hierarchy level and evaluate the robustness of each of these methods on different data sets. Finally we provide a framework for selecting the architecture the hierarchical GP for given hardware and discuss the implications of implementing our model in practice. Our practical framework allows the application of GPs to data sets with tens of millions of data points without and explicit sparse representation. Hence, our model substantially advances the current state of the art in large-scale GP modelling and finally makes it possible to apply non-parametric GPs to Big Data.

Acknowledgements

I would like to thank my parents and family for their continued support throughout the course of my education, which would not have been possible without the sacrifices they have made. I would also like to thank Rhea for her unwavering support and constant encouragement in my endeavours.

I would also like to thank Dr. William Knottenbelt for his generosity in granting me access to the CAMELOT cluster, Dr. Daniel Jones for his guidance through the early stages of my journey in machine learning, and Dr. Aldo Faisal for his time as a second marker and his lab (Andreas, Joanna) for providing this project with an interesting data set.

Most importantly, I would like to thank my supervisor, Dr. Marc Deisenroth for his excellent mentorship and support during the course of my project.

Contents

Chapter 1

Introduction

The Gaussian process (GP) [1] is an expressive and flexible model for machine learning. However, GPs are limited in practical applications by the size of the data set that they can be applied to, due to training that scales in $\mathcal{O}(N^3)$ and prediction that scales in $\mathcal{O}(N^2)$, where N is the number of points in the training data set. Sparse approximations [2, 3, 4, 5, 6, 7] exist, but are typically limited to $N < 10^6$ data points.

Machine learning techniques are integral to modern day 'big data' problems. With the ever increasing amount of data being collected from the internet, mobile device users and readings from ubuiquitous sensors across the globe, some of these methods, including the GP may fail to remain relevant, simply because the amount of computational power required to apply them to large data sets. If the size of the data restricts us to using only simpler and cheaper methods such as linear regression, but the data contains non-linear patterns, then we leave ourselves unable to maximise the valuable information the data can potentially provide.

We develop a hierarchical GP model, which can scale to $N > 10^7$ data points for training and predictions. The main features of this model are:

- Highly scalable approximation to a full Gaussian process regression model for large scale Bayesian inference.
- Enabling massively parallelisation of computations to be distributed over a large number of computational units (multi-core processors and distributed systems).
- Flexibility in applying to different types of data (densely clustered or sparsely scattered data).

These features allow us to apply GP modelling to large data sets on large modern day computing clusters, giving GPs a new lease of life in this modern era of machine learning.

Symbols and Notation

Throughout this paper, all vectors are represented with boldface lowercase symbols (e.g., $\mathbf{x}$) and matrices with bold uppercase symbols (e.g., $\mathbf{K}$) unless otherwise specified.

$\mathcal{G}$	Gaussian process
$\mathcal{H}$	Hierarchical Gaussian process
Θ	A set of hyperparameters
$(\mathbf{x})_i$	The i-th element of the vector $\mathbf{x}$
$\mathbf{x}^\top$	The transpose of the vector $\mathbf{x}$
$\mathbf{0}$	A column vector of zeros
$\text{diag}(\mathbf{x})$	A diagonal square matrix with the elements of the vector $\mathbf{x}$ along its main diagonal
$(\mathbf{A})_{ij}$	The i,j-th element of the matrix $\mathbf{A}$
$\lvert\mathbf{A}\rvert$	Determinant of the matrix $\mathbf{A}$
$\mathbf{A}^{-1}$	The inverse of the matrix $\mathbf{A}$
$\mathbf{A}^\top$	The transpose of the matrix $\mathbf{A}$
$\mathbf{I}$	The identity matrix
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	A Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
$\mathcal{N}(\cdot\lvert\boldsymbol{\mu}, \boldsymbol{\Sigma})$	The probability distribution function (pdf) of a Gaussian random variable with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Chapter 2

Background

2.1 Gaussian Processes for Regression

We have a training data set $\mathcal{D}$ consisting of N input vectors $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^N$ ($\mathbf{x}_i \in \mathbb{R}^d$ where d is the dimension of the input space) and corresponding noisy targets $\mathbf{y} = \{y_i\}_{i=1}^N$ ($y_i \in \mathbb{R}$) observed at each $\mathbf{x}_i$. The model for the noisy observations is

$$y_i = f(\mathbf{x}_i) + \epsilon_i \tag{2.1}$$

where $f(\cdot)$ is the underlying latent function and ϵ_i the independent noise variables in the observations ($\epsilon_i \sim \mathcal{N}\left(0, \sigma_n^2\right)$). In GP regression, we place a zero mean Gaussian prior on the underlying latent function. Therefore, any finite subset of latent variables will also have a (zero mean) multivariate Gaussian distribution as follows.

$$p(\mathbf{f}|\mathbf{X}) = \mathcal{N}\left(\mathbf{f}|\mathbf{0}, \mathbf{K}\right) \tag{2.2}$$

The elements of $\mathbf{K} \in \mathbb{R}^{N \times N}$ are given by the covariance function, or *kernel*, $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Here we have $(\mathbf{K})_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. The kernel encodes implicit assumptions about the underlying function $f(\cdot)$. These may include properties such as smoothness and periodicity. As such, the choice of $k(\cdot, \cdot)$ depends on the structure of the data and therefore changes with the problem at hand. A example is the Gaussian kernel.

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2l^2}\right) \tag{2.3}$$

The hyperparameters of the Gaussian kernel $\Theta = \{\sigma_f, l\}$. Given the latent function values, the likelihood of $\mathbf{y}$ is

$$p(\mathbf{y}|\mathbf{f}, \mathbf{X}) = \mathcal{N}\left(\mathbf{y}|\mathbf{f}, \sigma_n^2\mathbf{I}\right) \tag{2.4}$$

and we can integrate out the latent values and write the prior on $\mathbf{y}$ as

$$p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{f},\mathbf{X})p(\mathbf{f}|\mathbf{X})d\mathbf{f} \tag{2.5}$$

$$= \mathcal{N}\left(\mathbf{y}|\mathbf{0},\mathbf{K}+\sigma_n^2\mathbf{I}\right). \tag{2.6}$$

2.1.1 Prediction

Having observed the data points in $\mathcal{D}$, we can make predictions about y_* given a new input point $\mathbf{x}_*$ by obtaining a conditional distribution for y_* given $\mathbf{y}$. We rewrite (2.6) to incorporate the prediction data point.

$$\begin{pmatrix} \mathbf{y} \\ y_* \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K}' & \mathbf{k}_* \\ \mathbf{k}_*^\top & k'_{**} \end{pmatrix}\right) \tag{2.7}$$

$\mathbf{K}' = \mathbf{K}+\sigma_n^2\mathbf{I} \in \mathbb{R}^{N\times N}$, $\mathbf{k}_* = (k(\mathbf{x}_*,\mathbf{x}_1),\cdots,k(\mathbf{x}_*,\mathbf{x}_N))^\top \in \mathbb{R}^N$, $k'_{**} = k_{**}+\sigma_n^2 = k(\mathbf{x}_*,\mathbf{x}_*)+\sigma_n^2 \in \mathbb{R}$. From (2.7) we can obtain the predictive distribution for y_*.

$$y_* \mid \mathbf{y},\mathbf{X},\mathbf{x}_* \sim \mathcal{N}\left(\mathbf{k}_*^\top\mathbf{K}'^{-1}\mathbf{y}, k'_{**}-\mathbf{k}_*^\top\mathbf{K}'^{-1}\mathbf{k}_*\right), \tag{2.8}$$

2.1.2 Training

Training of a Gaussian process is done by selecting the set of parameters $\hat{\Theta}$ for the kernel which best explains the data. We do so by maximising the *marginal likelihood* of the training observations with respect to the parameters in Θ. For numerical stability, the log of the marginal likelihood is used instead.

$$\log p(\mathbf{y}|\mathbf{X};\Theta) = -\frac{1}{2}\mathbf{y}^\top\mathbf{K}'^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{K}'| - \frac{N}{2}\log 2\pi. \tag{2.9}$$

The marginal likelihood naturally trades off data fit and model complexity, i.e. it is a practical implementation of Occam's razor.

2.1.3 Problems

The main computational cost in training a GP arises from computing $\mathbf{K}'^{-1}$ and $|\mathbf{K}'|$, which are $\mathcal{O}(N^3)$ operations. Having obtained $\mathbf{K}'^{-1}$, computing the term $\mathbf{k}_*^\top\mathbf{K}'^{-1}\mathbf{k}_*$ for prediction requires performing an inner product with respect to the $N\times N$ matrix $\mathbf{K}'^{-1}$ which, scales in $\mathcal{O}(N^2)$. In practice, the terms are not computed explicity as per their expressions above (e.g., using left division instead of computing the inverse), but the computational complexity remains as stated.

2.2 Sparse Gaussian Process Models

Various sparse approximations have been proposed to scale GP modelling to larger data sets. These methods generally have the form of a GP model with a sparse set of data (which may or may not be drawn from the actual training data) and are trained by maximising the predictive likelihood of the actual training data. We discuss one representative method in the following section. Other approaches [2, 3, 4, 5, 6, 7] to sparse GPs are based on similar principles and a unifying overview of these methods is given in [8].

Sparse Pseudo-Inputs Gaussian Processes (SPGP)

Ghahramani and Snelson [9] present a computationally tractable method of implementing GPs using sparse pseudo-input Gaussian processes. Instead of training the GP on the entire data set of N data points, inference is done using a pseudo data set $\bar{\mathcal{D}} = \{\bar{\mathbf{X}}, \bar{\mathbf{f}}\}$ of size M where $M \ll N$. $\bar{\mathbf{f}}$ is used instead of $\bar{\mathbf{y}}$ here since the pseudo observations are assumed to have no noise. Given the pseudo data set, the likelihood of the actual training data is

$$\mathbf{y}|\bar{\mathbf{y}}, \bar{\mathbf{X}}, \mathbf{X} \sim \mathcal{N}\left(\mathbf{K}_{NM}\mathbf{K}_M^{-1}\bar{\mathbf{f}}, \mathbf{\Lambda} + \sigma_n^2\mathbf{I}\right) \tag{2.10}$$

where $\mathbf{\Lambda} = \text{diag}(\boldsymbol{\lambda})$, $(\boldsymbol{\lambda})_i = k_{ii} - \mathbf{k}_i^\top \mathbf{K}_M^{-1}\mathbf{k}_i$, $k_{ii} = k(\mathbf{x}_i, \mathbf{x}_i)$, $\mathbf{k}_i^\top = (k(\mathbf{x}_i, \bar{\mathbf{x}}_1), \cdots, k(\mathbf{x}_i, \bar{\mathbf{x}}_M))$, $(\mathbf{K}_{NM})_{ij} = k(\mathbf{x}_i, \bar{\mathbf{x}}_j)$ and $(\mathbf{K}_M)_{ij} = k(\bar{\mathbf{x}}_i, \bar{\mathbf{x}}_j)$.

This can be thought of as the probability of observing the actual training targets given a GP model trained with a smaller data set of M points. Equation (2.10) can be computed in $\mathcal{O}(M^2N)$ and can significantly reduce the cost of training a GP. The prediction cost for the variance is $\mathcal{O}(M^2)$.

In the SPGP model, the pseudo inputs can be treated as hyperparameters and be learnt in the same way that the kernel parameters are learnt. Their initial positions can be drawn from the actual training data set. This introduces Md dimensions to the hyperparameter search space and thus increases the complexity of the search for the global optimum. The choice of the size M of the pseudo training set is also an unclear one. Thus far, there has been no literature on selecting the appropriate size for the pseudo data set. This introduces a new dimension to the problem and may incur even larger costs if multiple trails have to be conducted to determine a suitable value of M to use for the problem at hand. More specifically, the effectiveness of the SPGP relies on the pseudo data set being able to 'explain' the structure of the actual data well enough, and choosing

the appropriate number of pseudo-inputs is not easily achievable, especially in high dimensional data, which is difficult to visualise.

2.3 Mixture of Gaussian Process Experts Models

Another variation applying the Gaussian process is by a mixture-of-experts (ME) model [10]. The ME model regards the underlying data generating process as a mixture of sub-processes defined on possibly overlapping regions of the input space. Each of these regions will be represented by a local expert model, and a *gating network* is required to determine the likelihood of each model given an input point. The gating network is a probability distribution over the local expert models given a particular input point, and it outputs a vector of likelihood values that can be thought of as the local models responsibilities towards the full likelihood. For each data point, the likelihood of the model is

$$p(y|\mathbf{x}) = \sum_i p(i|\mathbf{x})p(y|\mathbf{x}, i). \tag{2.11}$$

The $p(i|\mathbf{x})$ are the outputs of the gating network, and $p(y|\mathbf{x}, i)$ are the likelihood of each local expert model. The ME model allows for input dependent variation such as heteroscedasticity to be modelled since each local model has its own set of parameters. The ME model is trained by maximising the full likelihood with respect to the gating network parameters and each local expert model's parameters. Prediction is made by collecting the predictions of all local expert models, and weighing them using the responsibilities assigned by the gating network.

The ME model has been adapted to GP modelling [11, 12, 13] with Gaussian processes as the local models. With a stationary covariance function, a standard Gaussian process generalises the behaviour of the underlying latent function across different regions in the input space, which may be an inappropriate assumption in certain cases. In addition to modelling non-stationarity, the MGPE method can also be used to reduce the computational complexity, by limiting the maximum number of points at each local GP expert.

The mixture-of-Gaussian-Process-experts (MGPE) models requires the data to be partition. In [12] a Dirichlet process prior is placed over the partitioning. However, it is computationally intractable and requires Monte Carlo Markov Chain (MCMC) sampling to assign data points to each Gaussian process expert, a computationally demanding process.

2.4 Summary

The computational intractability of the Gaussian process on large data sets have led to developments of two types of methods that address this problem. The first is the sparse approximation, which uses a small subset of inducing inputs to model the underlying function. This relies on having enough inducing inputs to capture the peculiarities of the underlying function. The second is the local expert models method, which divides the input space into regions and fits small local models, which are computationally tractable. The success of these methods on large data sets raises two salient points. The validity of the sparse method suggests that the information about the underlying function can be captured using a small number of data points, and that additional data points provide only slightly more information. The local expert models method suggests that independent local models across different regions of the input space can sufficiently model the underlying function, since the behaviour of the underlying function may vary across the different regions. We will revisit these two themes and use them to develop a data selection method for our model.

Chapter 3

Hierarchical Gaussian Process

In GP regression as described in Section 2.1, we can think of the underlying latent function $f(\cdot)$ that we are trying to model as an infinite-dimensional Gaussian distributed random vector, which we make inference about based on a finite number of observations (as in (2.2)).

In statistical modelling the bootstrap method allows us to make inference on the properties of a population when the available sample size is insufficient by resampling from the sample and performing the inference on the resampled samples. We adopt this idea of making inference about a population by resampling a number of smaller samples and apply it to Gaussian process regression by treating the latent function as the population and our observations as the sample (although we deviate from the standard bootstrapping techniques along the way).

The computational intractability of the GP comes from inverting the $N \times N$ covariance matrix. We begin with the motivation of having a block diagonal approximation of the covariance matrix in our model to a series of $B \times B$ blocks, which lowers the computational cost of to $\mathcal{O}(B^3)$ (since the inverse of a block diagonal matrix is a block diagonal matrix with the inverse of each block on the diagonal). In the GP model, a block diagonal covariance matrix corresponds to the assumption of indepdences between data points belonging to subsets represented by different blocks in the block diagonal. This is not an assumption that can readily be made about the data in general, and we account for this independent assumption by having overlaps among the subsets that the block diagonals represent.

Consider a Gaussian process $\mathcal{G}_{\mathcal{D},\Theta}$ with the training data set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$. We define a set of c subsets of the data set $\mathcal{D}$ as $\mathcal{S} = \{\mathcal{D}^{(1)}, \cdots, \mathcal{D}^{(c)}\}$ where $\mathcal{D}^{(i)} = \{\mathbf{X}^{(i)}, \mathbf{y}^{(i)}\}$. These subsets of $\mathcal{D}$ correspond to the bootstrap samples of the original training set and will use a GP on each of them. We will use these GPs to make our inference of the full GP $\mathcal{G}_{\mathcal{D},\Theta}$. In our model, all the GPs

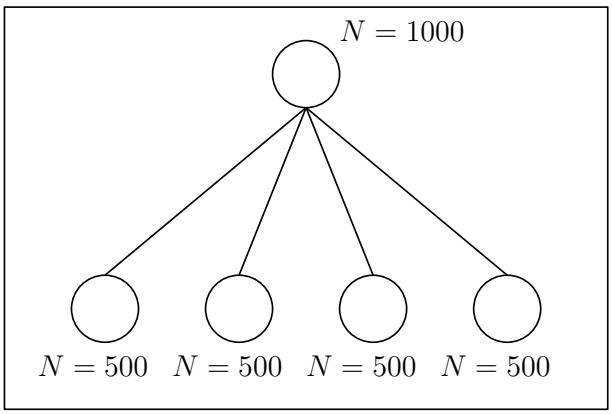

Figure 3.1: Structure of a HGP. In this example, the full training data set is resampled four times, each time with 500 observations. Each resample is handled using a GPs.

will be jointly trained and they share a single set of parameters (this applies for both training and prediction and will be detailed in the following sections).

In a typical bootstrapping experiment, from the original sample of N data points we construct the resamples (each also of size N) by sampling from the original sample with replacement. In our model, however, we do not adhere to this convention, and instead use a smaller resample size. The motivation behind this is that we want each GP to use a smaller subset of the full data set and, therefore, be computationally cheaper to handle. We exploit this property to parallelise and distribute computations over independent processing units.

This method of creating subsets of the training data and using each subset with a GP forms the basis of the hierarchical Gaussian process (HGP) model, where we have divided the problem into a number of Gaussian processes each using a smaller training data set. This can be done recursively, using a HGP at each child-GP and further subdividing the problem until a desired size[1] is achieved.

[1]This may be chosen based on criteria such as computational time restrictions or available hardware.

3.1 Definitions

Where a HGP is used in place of a standard GP, we write $\mathcal{H}$ instead of $\mathcal{G}$. We adopt the following terms and notation in this paper

Child-GP	A child-GP of a HGP is a GP that is used with a subset of the training data. A child-GP may also be a HGP in the recursive case.
Parent-GP	The converse notion of child-GP. $\mathcal{P}$ is the parent of $\mathcal{C}$ if and only if $\mathcal{C}$ is a child of $\mathcal{P}$. If $\mathcal{C}_1, \cdots, \mathcal{C}_c$ are the c child-GPs of $\mathcal{G}$ we write $\mathcal{G} = \{\mathcal{C}_1, \cdots, \mathcal{C}_c\}$ and $\mathcal{C}_i \in \mathcal{G}$.
Sibling	Two GPs/HGPs are siblings if and only if they share the same parent.
Root	The root of a HGP $\mathcal{H}$ is the top most HGP in $\mathcal{H}$. It is denoted by root($\mathcal{H}$). We refer to the root of a HGP as level 0.
Leaf	A leaf of a HGP is a standard GP at the bottom of the HGP tree. A leaf-GP has no children. leaves($\mathcal{H}$) denotes the set of all the leaf-GPs of $\mathcal{H}$. leaves($\mathcal{H}$) $= \bigcup_{\mathcal{H}' \in \mathcal{H}}$ leaves($\mathcal{H}'$). We refer to the leaf-GPs of a HGP with l levels as level l.
Size	The size of a HGP is the number of child-GPs it has. size($\mathcal{H}$) denotes the size of $\mathcal{H}$.
Depth	The depth of a HGP $\mathcal{H}$ is a positive integer that indicates the path length from the root of $\mathcal{H}$ and its bottommost leaf. depth($\mathcal{H}$) denotes the depth of $\mathcal{H}$. Note that if $\mathcal{H}$ is a standard GP, then depth($\mathcal{H}$) $= 0$.

Definition 1. *A HGP $\mathcal{H}$ (whose depth is greater than one) is* balanced *if and only if for every pair of sibling HGPs $\mathcal{D}_1, \mathcal{D}_2$ in $\mathcal{H}$, size($\mathcal{D}_1$) = size($\mathcal{D}_2$).*

HGP Architecture Notation

For a balanced HGP $\mathcal{H}$ with depth($\mathcal{H}$) $= l$, we use an ordered set of l integers to denote its structure: $(c_0, \ldots, c_{l-1})$. c_i in this vector represents the number of children a HGP at level i. See Figure 3.2.

3.2 Training

One way of training a GP is to select the parameters $\hat{\Theta} = \text{argmax}_\Theta \log p(\mathbf{y}|\mathbf{X}, \Theta)$ by maximising the log marginal likelihood (Equation 2.9 in Chapter 2.1. Typically, when the bootstrap method

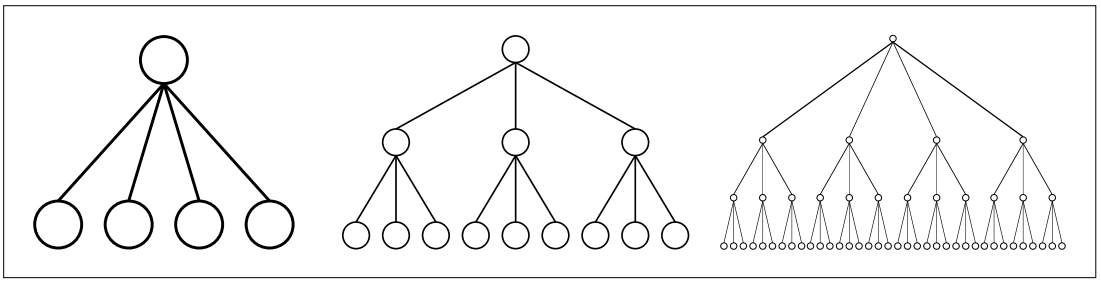

Figure 3.2: HGP architecture notation (from left to right): (4), (3,3), (4,3,3)

is used, parameter estimation is done by *bootstrap aggregating*, in which the desired parameter is computed from each sub-sample, and their outputs are averaged to produce the final output.

An obvious way of applying this idea to the HGP model is to train each child-GP separately to obtain a set of parameters from each sub-sample and take the average of these parameters. However, a potential drawback of doing this is that the marginal likelihood is not linear in the kernel parameters (for example, the lengthscale parameter in the Gaussian kernel in (2.3)). Thus taking the arithmetic mean of the parameters obtained from training the child-GPs on different sub-samples results causes inaccuracies. We illustrate this with an extreme example. Suppose the lengthscale parameter in the Gaussian kernel for one of the child-GPs is large enough that the argument of the exponential term is effectively zero. Then, increasing the lengthscale parameter for this particular child-GP would have no effect on the marginal likelihood of the child-GP, but it would affect the average of the parameters (if, for the sake of this illustration, we assume that the lengthscale parameters from the other child-GPs are relatively small). This would lead to poor results and we propose an alternative method of training the HGP.

Since the data in each each child-GP is drawn independently from the full data set, we consider the child-GP to be independent, and thus the marginal likelihood for the HGP is given as follows.

$$p(\mathbf{y}|\mathbf{X},\Theta) \propto \prod_{i=1}^{c} p(\mathbf{y}^{(i)}|\mathbf{X}^{(i)},\Theta) \tag{3.1a}$$

$$\log p(\mathbf{y}|\mathbf{X},\Theta) = \sum_{i=1}^{c} \log p(\mathbf{y}^{(i)}|\mathbf{X}^{(i)},\Theta) + \text{constant} \tag{3.1b}$$

The above equations are approximations to the full GP marginal likelihood which results from our assumption that the child-GPs are independent. Since $p(\mathbf{y}|\mathbf{X},\Theta)$ is proportional to the product of likelihoods, we get an additional constant term independent of $\boldsymbol{\Theta}$ (3.1b). Since we are only

interested in the relative value of $p(\mathbf{y}|\mathbf{X}, \Theta)$ for optimisation, we can ignore this term. Training the HGP by maximising this quantity instead of parameter selection by averaging the parameters found by separately training each child-GP, we are able to represent the contribution of each child-GP to the parent equally and we can better capture the effects of adjusting the parameters to the full model. Where the child-GP is a HGP itself, then each expression in the product in (3.1b) can be computed recursively. Otherwise, the expression in (2.9) for the log marginal likelihood of a full GP is used for the leaf-GP operating on a small subset of the data.

3.3 Prediction

The idea of bootstrap aggregating naturally leads to taking the average of the prediction outputs from each of the child-GPs as the HGP model's prediction. However, we can derive a predictive distribution of the HGP from a probabilistic approach instead. In (2.8) we see that the prediction of the GP model comes in the form of a Gaussian distribution.

As we did in (3.1) for the likelihood of the HGP, we derive a similar expression for the predictive distribution for the HGP. Let $\mathcal{N}\left(\mu_i, \sigma_i^2\right)$ be the predictive distribution of the i-th child-GP (denoted $\mathcal{C}_i$).

$$p(y_*|\mathbf{x}_*, \mathcal{H}) \quad \propto \quad \prod_{i=1}^{c} p(y_*|\mathbf{x}_*, \mathcal{C}_i) \tag{3.2}$$

Since (3.2) this is the product of Gaussian likelihoods, the predictive distribution of $\mathcal{H}$ is $\mathcal{N}\left(\mu_{\mathcal{H}}, \sigma_{\mathcal{H}}^2\right)$ where

$$\mu_{\mathcal{H}} = \sum_{i=1}^{c} w_i \mu_i \,, \qquad \sigma_{\mathcal{H}}^2 = \sum_{i=1}^{c} w_i \sigma_i^2$$
$$w_i = \sigma_i^{-2} \left(\sum_{j=1}^{c} \sigma_j^{-2} \right)^{-1} .$$

This can be thought of as a weighted average of the predictive mean and variance of each child-GP, with the corresponding weight w_i inversely proportional to the child's predictive variance. This is expected, as the predictive variance gives us a measure of our confidence in the prediction, therefore having the inverse as weights allows the more likely predictions having a larger contribution to the final predictive output.

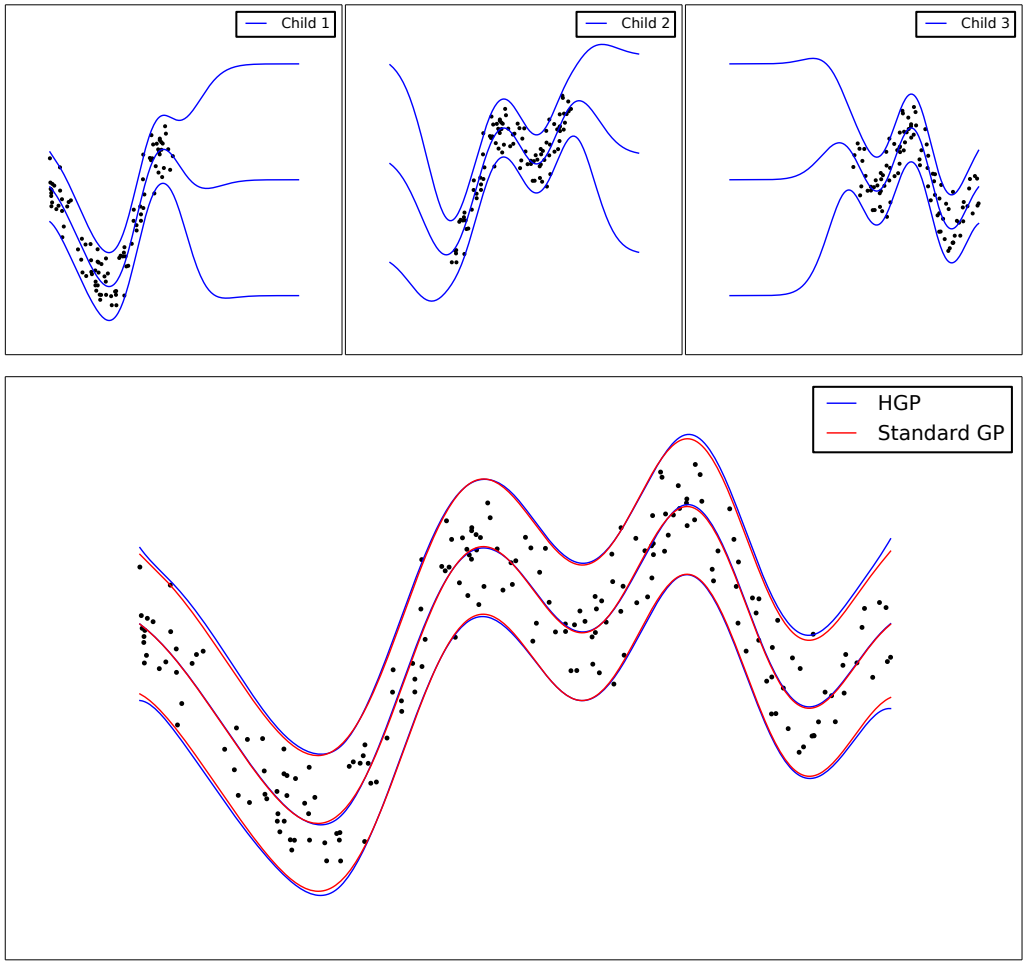

Figure 3.3: Top: the red lines represent the predictive mean and variance (plotted as two standard deviations above and below the mean) of each child-GP. Regions with no input points correspond to high predictive variances. Bottom: the red lines show the predictive output of the parent GP (whose child-GPs predictive outputs are shown in Figure 3.3) compared against the full GP in blue.

3.4 Illustration

We apply the HGP model to a toy data set of 200 training points with a HGP with 3 child-GPs (Figures 3.3). Each child-GP is given a subset of 80 training points. We have deliberately selected the training points such that each child-GP receives points within a contiguous segment of the input space to illustrate the HGP's ability to extract the relevant predictions from the child-GPs into the final prediction. In the child-GP predictions we see that the locations where the inaccurate predictions are made correspond to higher predictive variances, and this reduces their weight in the HGP's prediction.

Chapter 4

HGP Architecture

We introduced the HGP model as an application of the bootstrap technique to the training data set of the full GP and 'resampled' it into a number of subsets from which we construct the child-GPs. The generic way of selecting the number of bootstrap samples is to increase it until certain convergence criteria are met. This suggests a large number of bootstrap samples, and in our model, a large number of child-GPs. This practice, however, is not suitable for the HGP, since the computational requirement of the HGP is linear in the number of child-GPs, and the computational cost at each child-GP is $\mathcal{O}(M^3)$ for training and $\mathcal{O}(M^2)$ for prediction where M is the size of the subset (resample). Moreover, we have also demonstrated in the toy example (Chapter 3.4) that with a small number of child-GPs we can recover the performance of a standard GP on the full data set. In the following sections we outline a few methods for subsetting the data for the child-GPs and discuss the implications of the design choices.

We train the HGP and make predictions with the assumption that each child-GP is independent of the other child-GPs. Suppose $\mathcal{S}$ is a partition of $\mathcal{D}$. This would regard the data in each subset as independent of the data in every other subset, hence, losing some information about the structure of the data (this information would otherwise be captured by the covariance terms in the full GP). For example, a certain set of parameters may maximise the likelihood of the child-GPs within the subsets, but not the likelihood of the full model. If, say, $\mathcal{S}$ is constructed based on a contiguous partitioning of the input space and a Gaussian kernel is used, maximising the likelihood of this model will result in a smaller lengthscale parameter, such that our model being cannot capture longer range dependencies of the data.

The mixture of experts (ME) model is applied with each expert assigned a different region of the input space to model input dependent variations such as heteroscedasticity as well as to divide

the data set into smaller subsets for computational tractability. The objective of our model differs in that we aim to replicate a standard GP and, therefore, we are interested in mitigate the effects of the independent assumption as described in the previous paragraph. We do so by having each of the subsets in $\mathcal{S}$ overlap therefore allowing the individual child-GPs to account for the covariance effects between the points in $\mathcal{D}$.

We evaluated the negative log marginal likelihood of three GP models: 1. a HGP with its child-GPs not sharing any data points, 2. a HGP with its child-GPs having overlapping subsets of the data, 3. a full GP. Figure 4.1 shows the plot of the negative log marginal likelihood with respect to the lengthscale parameters. The data used was generated from a zero mean GP prior with a lengthscale of 2. All other parameters were fixed for the data generation and marginal likelihood computations. The negative log marginal likelihood of the HGP with non-overlapping child-GPs is minimised at a smaller lengthscale of 1.77, compared to a HGP with overlapping partitions (lengthscale minimises negative log marginal likelihood at 1.85), followed by that of a full GP (minimised at 2.02). This demonstrates the effects of assuming independence between subsets.

4.1 Child-GP Construction

We begin by partitioning the data set $\mathcal{D}$ into $\mathcal{P}$, a set of p disjoint subsets, and select combinations of subsets for each child-GP to create overlaps amongst child-GPs.

4.1.1 Partitioning

P1 **Input space partitioning**

The input space is divided into non-overlapping regions, and each subset is assigned one region. Depending on how the data points are distributed, this may result in non-uniformity in the sizes of the subsets. To deal with this, we use the kd-tree algorithm. A kd-tree is a tree data structure in which keys have multiple dimensions. At each node, the keys is split into two subsets (into the left and right subtree) at the median value of the largest dimension. This is done recursively until each node contains a single key. For our purposes we do not require the data to be divided all the way into the leaf nodes, and we can terminate the division when the required number of partitions is reached. It should be noted that the number of partitions obtained from this algorithm will always be a power of 2. We refer to this method

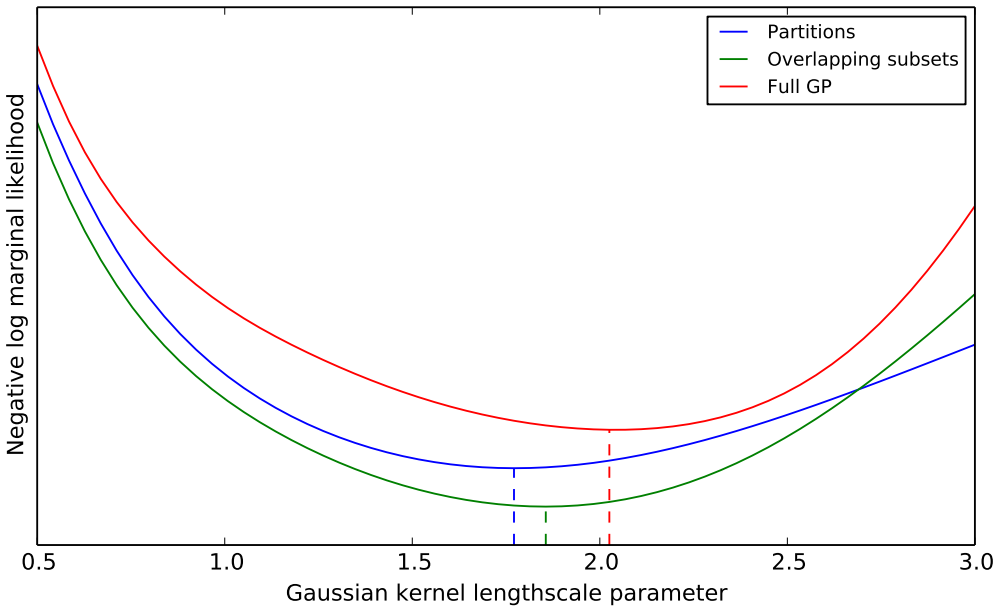

Figure 4.1: Plots of the negative log marginal likelihood with respect to the lengthscale parameter of the Gaussian kernel for 1. a HGP with child-GPs constructed on a partition of the full data set, 2. a HGP with child-GPs that have overlapping data sets, 3. a full GP. Note that the absolute y values of these plots have no significance (values have been scaled to plot the curves on the same figure).

as the 'clustering method' since the distribution of points within each partition created by this method is clustered into a region of the input space.

P2 **Uniform over input space**

Method P1 is first applied and a partition of points by segmenting the input space (call this partitioning $\mathcal{Z}$). Each subset in $\mathcal{Z}$ is then partitioned into p disjoint groups (we avoid the use of 'subset' again here to prevent confusion). We then construct each subset in $\mathcal{P}$ by uniquely selecting *one* group from each subset in $\mathcal{Z}$. This method is a contrast to method P1, in that each subset will contain points across the input space, rather than being clustered in the same region in the input space. We refer to this method as the 'scattering' method since the distribution of points within each partition created by this method is scattered over the input space.

P3 **Random selection**

Given p partitions, each point in the training set is assigned to a unique partition. The random assignment is generated uniformly over the integers $1, \ldots, p$, and generally this will result in each subset containing a sparse collection of data points across the input space covered by the main training data set, as described in method P2, but is much more cost effective to perform for large data sets, since it can be done without having to access the data at the time of building the HGP.

We illustrate methods P1 and P2 in Figure 4.2. A synthetic two-dimensional input space was generated and the data was divided into four subsets. The clustering (P1) and scattering (P2) methods have been designed with two purposes in mind. The motivation behind the HGP is a model that is able to handle large data sets. A large data set may generally be classified into one of the following two types:

- **Dense**

  A data set is considered dense if, for a given region in the input space, there is a relatively large number of observations.

- **Sparse**

  A data set is considered sparse if, for a given region in the input space, there is a relatively small number of observations.

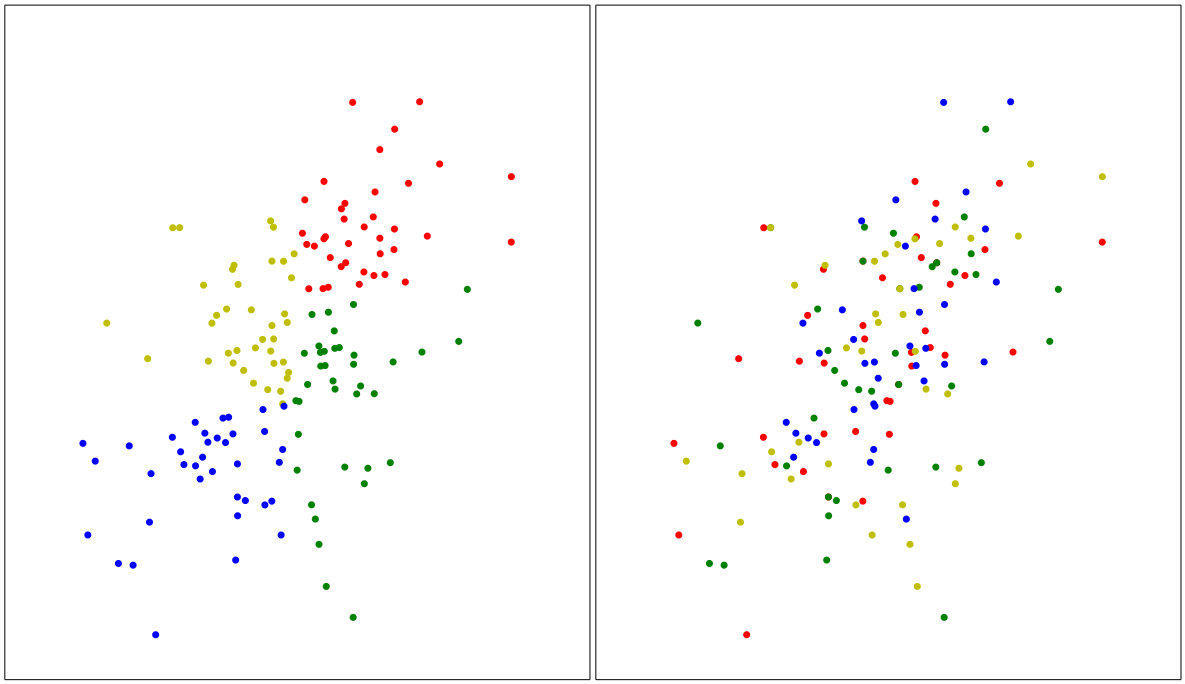

Figure 4.2: The panel on the left shows the result of using method P1 (input space partitioning) on the input data, while the panel on the right shows the result of using method P2 (uniform over input space). The colour of the point indicates membership in a subset.

Note that the terms *dense* and *sparse* used in this context, are relative. This is demonstrated in Figure 4.3. In the first panel, we have a data set of 100 points drawn from a certain region of the input space. In the second, we have 300 points drawn from the same region of the input space (dense), and in the third we have 300 points from a larger region of the input space (sparse).

In the dense case, a small subset of the data uniformly drawn can give us sufficient information about the underlying function (as seen in Panel 1), and therefore the large amount of data available provides us only with marginally more information. In other words, the data we have is highly repeated. In situations such as this, the scattering method is appropriate, since each child-GP have enough data to model the underlying function, and it is likely that each child-GP will have similar prediction weights. This works to the effect of having many sub-models, each on a small set of observed data, and averaging the models.

In the sparse case, the large amount of data we have (in addition to the data already present in the original data set) contains new information about the underlying function in a different region of the input space. If we use the scattering method here, each partition created will contain much less data per region of the input space, which may lead to each child-GP not being able to model the

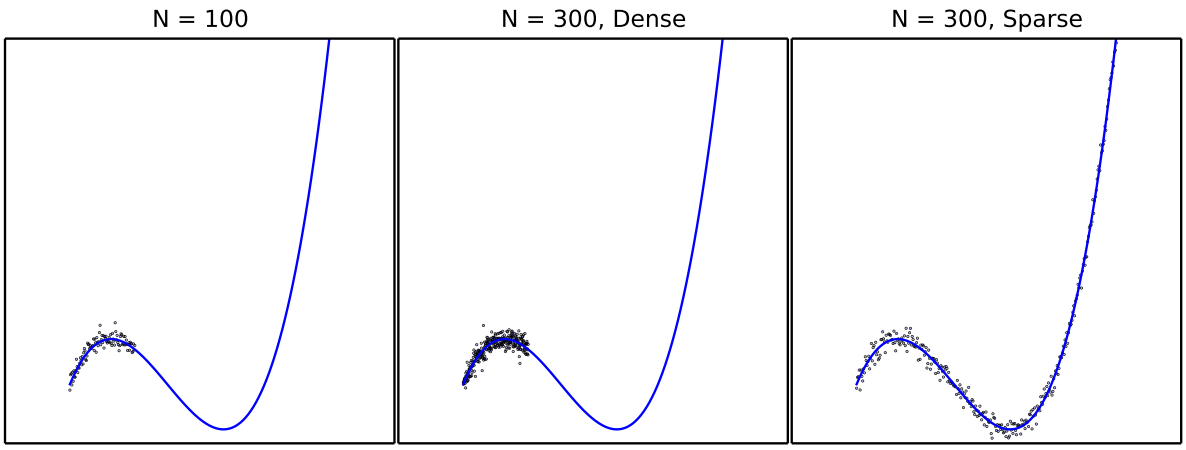

Figure 4.3: The first panel depicts a data set of 100 samples drawn from one region in the input space. The second panel depicts a large and 'dense' data set of 300 samples drawn from the same region and the third panel depicts a large and 'sparse' data set of 300 samples from a larger region in the input space.

underlying function with sufficient resolution. Instead, the clustering partitioning method allows us to have each child-GP having enough points within regions of the input space in order to model variations of the underlying function within smaller changes in the input space. The combination of partitions (not neccesarily next to each other) in each child-GP also allows us to capture the longer range variations of the underlying function. This method approximately works to the effect of a mixture-of-local-experts model, since each child-GP sees groups of data local to small regions in the input space.

4.1.2 Combining Partitions

C1 **All combinations of pairs of subsets**

This is done by selecting all possible pairs of subsets in $\mathcal{P}$. This will result in $\binom{p}{2} = \frac{p(p-1)}{2}$ child-GPs. This selection ensures that the covariances between every pair of points in distinct subsets are computed.

C2 **All possible combinations of m subsets**

Similar to the previous method. We pick all possible combinations of m subsets, which results in $\binom{p}{m}$ child-GPs. This selection also ensures that the covariances between every pair of points in distinct subsets are computed. The amount of overlap between child-GPs increases with m, at the cost of having more child-GPs.

C3 **Representation criterion**

We fix an integer value r and assign subsets to child-GPs such that each subset appears in r child-GPs, and each child-GP does not get assigned the same subset more than once. $r = 1$ corresponds to having non-overlapping child-GPs, and $r = c$ (where c is the number of child-GPs) corresponds to having the full GP repeated c times. Therefore, the condition $1 < r < c$ should be imposed. r determines the amount of overlap between child-GPs without having affecting the number of child-GPs (we first set the number of child-GPs before deciding on r).

In this paper and our experiments, we will focus on method C3 since it has shown to be effective[1], and comes at a lower computational cost than C1 and C2.

4.2 Architecture Selection

4.2.1 Number of Child-GPs

Computational Complexity

In the previous section, we outlined a number of methods to select partition sizes and to combine these partitions to form the data sets for the child-GPs of a HGP. The time complexity for training and prediction for a HGP with c children and M training data points in each child are $\mathcal{O}(cM^3)$ and $\mathcal{O}(cM^2)$ respectively. If we allow M and c to vary inversely proportionately (i.e. $M \propto N/c$, which is the case if method C3 is used) and we choose the training set size of each child-GP based on the number of child-GPs, then the time complexity with respect to c is $\mathcal{O}(N^3/c^2)$ and $\mathcal{O}(N^2/c)$. Thus, for any N, to minimise the computational costs, we should opt for a large number of child-GPs. While this is an attractive proposition, there is a drawback of having a large number of child-GPs, which we discuss next.

Accuracy

Figure 4.4 (top row) illustrates the effect of the number of child-GPs on the accuracy of the HGP. Using method P2 and C3 (with $r = 2$), we constructed 3 HGPs with 4, 16, and 64 children on a training data set of size 200. This resulted in each HGP having child-GPs of sizes 100, 25, and 6 respectively (up to differences resulting from integer division remainders). As the number of

[1] Experiment results with HGPs constructed with method C3 in Figure 4.6 and 4.7 show this.

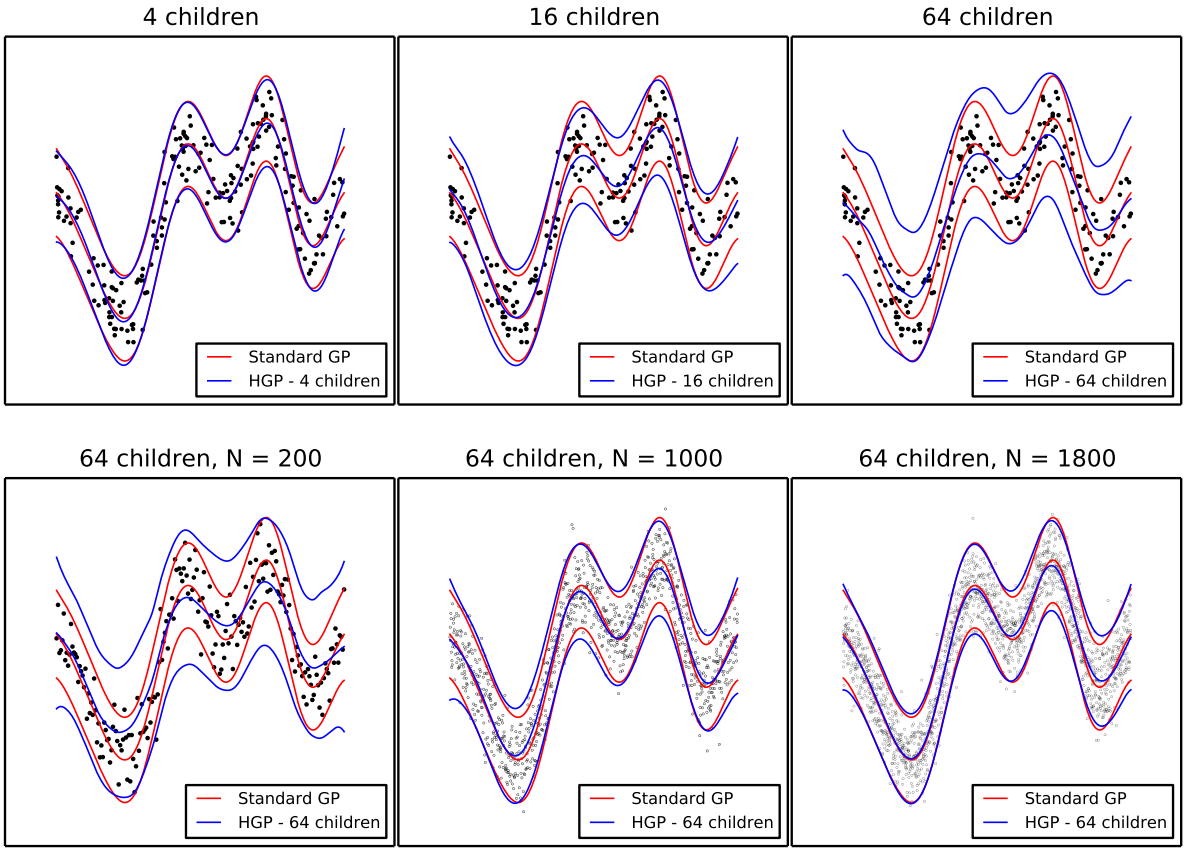

Figure 4.4: Top row: comparison of depth-1 HGPs (blue lines) with varying number of child-GPs against a standard GP (red lines). Prediction made with the hyperparameters found from training a standard GP (by maximising the marginal likelihood). The prediction accuracy decreases with the number of child-GPs. Bottom row: a HGP with 64 children making predictions using training sets of different sizes. With a larger training data set size, the loss of accuracy from having more child-GPs is recovered.

child-GPs increases, the accuracy decreases, as demonstrated by the HGPs with 16 child-GPs and 64 child-GPs. However, as we generate additional training samples from the (full) GP posterior distribution, we see that a HGP with 64 child-GPs is able to recover the prediction accuracy of the full GP. In the case of 200 training samples and 64 child-GPs, each child-GP does not have enough data points to model the underlying latent function, especially variations across smaller regions in the input space (as shown by the 'flatter' predictive output). With a larger training data set generated, each child-GP receives a larger number of data points, which contains more information about the underlying function. We see that this results in the HGP regaining the predictive accuracy of a full GP.

Design Choice

The above suggests that the choice of number of child-GPs is data-dependent. If we have a large set of training data that is dense[2] in the the input space (as we have shown in Figure 4.4 by generating new samples over the same domain), then we can afford to have a large number of child-GPs since each child-GP will be able to capture a sufficient amount of information to model the underlying function well. If, however, the training points are sparse[2] in the input space, then having a large number of child-GPs will result in poor performance.

4.2.2 Depth

So far we have only looked at HGPs that are one level deep. The HGP model can also be used recursively to create a tree structure. In the previous section, we demonstrated that while a large number of child-GPs may result in lower computational costs, there is also a loss of accuracy associated with it. In this section, we explore the possibilities of deep HGPs as an alternative. We will focus on balanced HGPs.

Computational Complexity

We have previously stated the time complexity of a one-level HGP as $\mathcal{O}(cM^3)$ for training and $\mathcal{O}(cM^2)$ for prediction. This generalises to $\mathcal{O}(cT)$ where $T \in \{T_{train}, T_{predict}\}$ is the time complexity at the leaf-GP required for training/prediction. A balanced HGP with depth v in which nodes at level l have c_l child-GPs will have $\mathcal{O}(T\prod_{l=0}^{v-1} c_l)$. The number of leaf-GPs in a HGP is $\text{leaves}(\mathcal{H}) = \prod_{l=0}^{v-1} c_l$, therefore, the time complexity of a multi-level HGP is proportional to the number of leaf-GPs it has (up to the size of the leaf-GPs).

If M_l is the size of the training data set of a HGP at level l, and we let the training set size of a child-GP be proportional to the number of children its parent has, then we have $M_l \propto M_{l-1}/c_{l-1}$.

[2]By dense (sparse), we refer to the property of, given a region in the input space, there data that there are many (few) data points.

By induction, we can derive the size of the training data set at the leaves.

$$\begin{aligned} M_0 &= N \\ M_1 &\propto M_0/c_0 \propto N/c_0 \\ M_2 &\propto M_1/c_1 \propto N/c_0c_1 \\ &\vdots \\ M_v &\propto M_{v-1}/c_{v-1} \propto N/c_0c_1\cdots c_{v-1} \propto N/\text{leaves}(\mathcal{H}) \end{aligned}$$

Therefore, $T_{train} \propto (N/\text{leaves}(\mathcal{H}))^3$ and $T_{predict} \propto (N/\text{leaves}(\mathcal{H}))^2$ and we arrive at the time complexity of the multi-level HGP of $\mathcal{O}(N^3/\text{leaves}(\mathcal{H})^2)$ for training and $\mathcal{O}(N^2/\text{leaves}(\mathcal{H}))$ for prediction. These expressions are consistent with those that we have earlier derived for a 1 level HGP. We also find that the asymptotic time complexity of the computation varying inversely with the number of child-GPs.

Accuracy

We perform the same comparison (Figure 4.5) of HGPs with varying depths (1,2,3) againsts a full GP (as we did for a one-level HGP previously). Similar observations were made: as the number of levels (hence number of leaves) increased, the prediction accuracy of the HGP (with respect to that of the full GP) deteriorated. The loss of accuracy is recovered as the density of the data was increased. Visually, for the same number of leaves, the output of a deep HGP (many levels) is more similar to output of a full GP than that of a one-level HGP (henceforth, in this context, which we shall refer to one-level HGP as a *wide* HGP to contrast it against a HGP with multiple levels). We analyse more thoroughly in the next section to compare the results of each type of HGP structure.

4.2.3 Analysis

We generate synthetic data from a GP prior to compare the accuracy of a wide HGPs and deep HGPs. We measure accuracy by comparing the predictive mean of each HGP model against a full GP and take the mean likelihood ratio (see Appendix A) for 1000 points uniformly spaced across the domain of the input space within which the training data falls in. We generate different data sets by varying the following.

- **Data density**

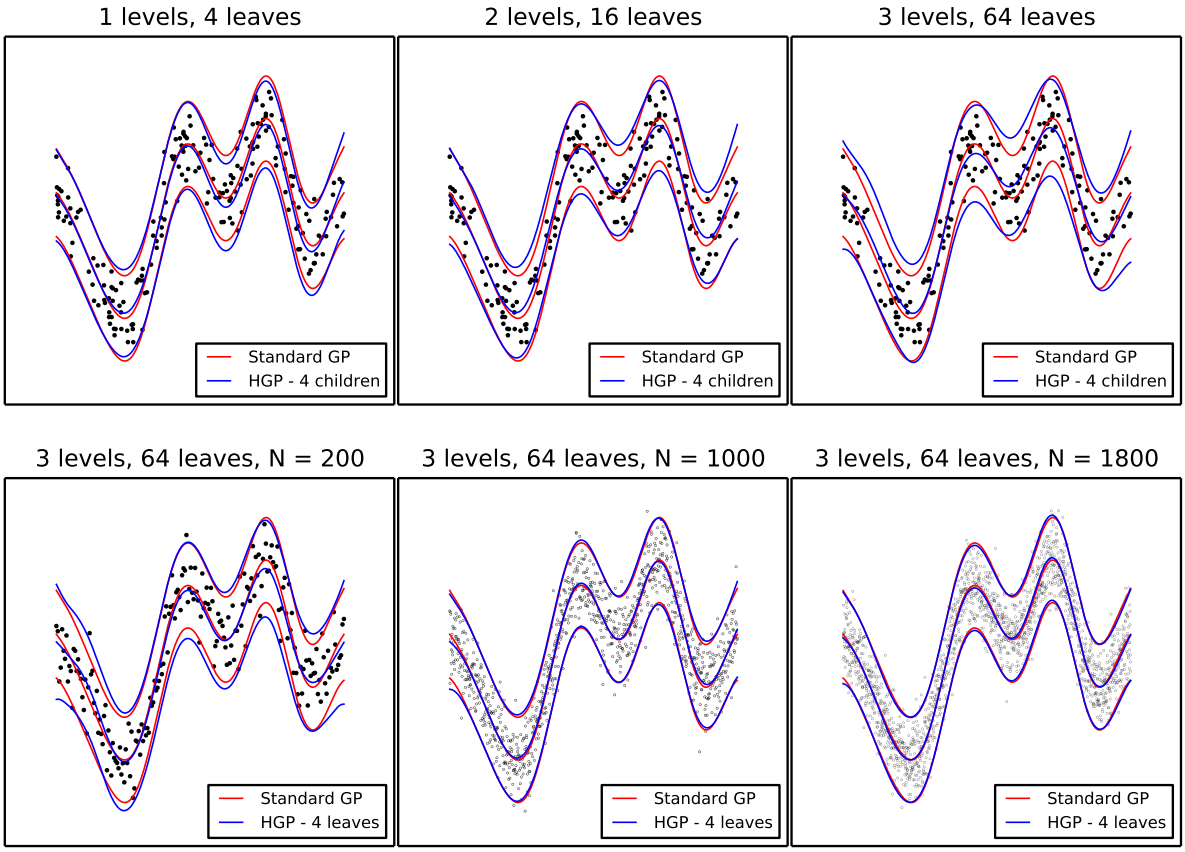

Figure 4.5: Top row: comparison of HGPs (blue lines) with varying depths against a standard GP (red lines). Prediction made with the hyperparameters found from training a standard GP (by maximising the marginal likelihood).

To achieve this, we fix the range within which the data falls in the input space and vary the data set size. Thus, for any region in the input space, we can expect a larger number of data points for a larger training set size.

- **Kernel parameters**

  The kernel encodes properties about the underlying latent function. Therefore, by generating data from GP priors with kernels using different parameters we can observe how our models perform across different function structures.

We fix the noise variance at 1, and make predictions using the GPs and HGPs with the same parameters that we generate the data from. Due to having randomly generated data, we run each experiment five times and average the results. The results are shown.

As we have seen in Figure 4.4 and 4.5, as the number of leaves of the HGP increase, the likelihood ratio of the HGP to the full GP falls. This is true for both wide and deep architectures.

We also observe across the variety of data we ran the tests on, that for a large number of leaves, a deep HGP behaves more similarly to a full GP than a wide GP does. Otherwise, we do not observe any significant difference in the performance of the HGP (in relation to a full GP) when the clustering or scattering method is used for the initial partition. The tables on the following two pages list the results.

Training Set Size	**Kernel Parameters**		**4 leaves**		**16 leaves**		**64 leaves**	
	Lengthscale	Size	Deep	Wide	Deep	Wide	Deep	Wide
400	2.0	2.0	0.9974	0.9969	0.9936	0.9902	0.9924	0.9735
		4.0	0.9972	0.9961	0.9936	0.9929	0.9941	0.9807
	4.0	2.0	0.9968	0.9978	0.9969	0.9974	0.9974	0.9858
		4.0	0.9978	0.9978	0.9892	0.9953	0.9957	0.9856
	8.0	2.0	0.9989	0.9991	0.9974	0.9971	0.9979	0.9875
		4.0	0.9967	0.9978	0.9982	0.9973	0.9974	0.9776
800	2.0	2.0	0.9980	0.9987	0.9977	0.9972	0.9966	0.9883
		4.0	0.9983	0.9979	0.9980	0.9976	0.9962	0.9897
	4.0	2.0	0.9980	0.9975	0.9979	0.9981	0.9956	0.9934
		4.0	0.9985	0.9984	0.9972	0.9982	0.9979	0.9924
	8.0	2.0	0.9991	0.9991	0.9993	0.9988	0.9970	0.9897
		4.0	0.9994	0.9991	0.9988	0.9988	0.9977	0.9942
1600	2.0	2.0	0.9991	0.9992	0.9980	0.9988	0.9976	0.9957
		4.0	0.9991	0.9991	0.9984	0.9987	0.9983	0.9944
	4.0	2.0	0.9991	0.9993	0.9987	0.9985	0.9986	0.9975
		4.0	0.9993	0.9992	0.9988	0.9991	0.9989	0.9962
	8.0	2.0	0.9996	0.9993	0.9996	0.9994	0.9990	0.9974
		4.0	0.9993	0.9994	0.9983	0.9991	0.9991	0.9984

Figure 4.6: Mean likelihood ratio of each HGP model (child-GP subsets constructed using the **scattering** method) to a standard GP model across different data densities and kernel parameters.

Training Set Size	**Kernel Parameters**		**4 leaves**		**16 leaves**		**64 leaves**	
	Lengthscale	Size	Deep	Wide	Deep	Wide	Deep	Wide
400	2.0	2.0	0.9988	0.9989	0.9979	0.9934	0.9930	0.9701
		4.0	0.9985	0.9983	0.9970	0.9942	0.9938	0.9691
	4.0	2.0	0.9997	0.9996	0.9976	0.9921	0.9946	0.9741
		4.0	0.9992	0.9990	0.9989	0.9980	0.9965	0.9872
	8.0	2.0	0.9997	0.9999	0.9990	0.9975	0.9981	0.9904
		4.0	0.9998	0.9998	0.9985	0.9971	0.9954	0.9791
800	2.0	2.0	0.9995	0.9995	0.9990	0.9968	0.9973	0.9855
		4.0	0.9994	0.9991	0.9984	0.9962	0.9961	0.9821
	4.0	2.0	0.9998	0.9997	0.9994	0.9984	0.9987	0.9942
		4.0	0.9998	0.9997	0.9992	0.9985	0.9975	0.9901
	8.0	2.0	0.9998	0.9999	0.9996	0.9991	0.9986	0.9930
		4.0	0.9999	0.9999	0.9997	0.9991	0.9994	0.9970
1600	2.0	2.0	0.9997	0.9997	0.9993	0.9983	0.9985	0.9922
		4.0	0.9998	0.9999	0.9991	0.9983	0.9983	0.9917
	4.0	2.0	0.9998	0.9998	0.9997	0.9992	0.9991	0.9952
		4.0	0.9998	0.9998	0.9996	0.9994	0.9988	0.9964
	8.0	2.0	1.0000	1.0000	0.9998	0.9995	0.9989	0.9938
		4.0	0.9999	0.9999	0.9998	0.9997	0.9994	0.9977

Figure 4.7: Mean likelihood ratio of each HGP model (child-GP subsets constructed using the **clustering** method) to a standard GP model across different data densities and kernel parameters.

4.3 Distributed Hierarchical GPs

We have seen in (3.1a) that the likelihood of the HGP is expressed as a product of independent terms. In practice we use the log of the marginal likelihood which is a sum of independent terms which can be computed separately. The same can be done for the derviatives with respect to the kernel hyperparameters, which are also sums of independent terms. Likewise, the prediction of a HGP (3.2) is a weighted sum of independent components. This allows us to perform the required computations for both training and prediction on independent processing units. Combining the results of the sub-computations is done at the parent which is $\mathcal{O}(c)$ where c is the number of child-GPs). This is insignificant to the main computational work (at the leaves, $\mathcal{O}(N^2)$ and $\mathcal{O}(N^3)$) and therefore we do not focus on the impact of these computations in our design.

Local Computation

Distributing the computational work of a HGP on a single shared memory space is straightforward. Since the computations in each child-GPs are independent, each child-GP is handled as one thread. Thread scheduling and load balancing is then left to the operating system to manage.

Distributed Computation

A distributed system consists of multiple computers connected over a network that communicate via message passing. For local computations, communication between separate threads (of the same process) is done via shared memory, which in general, is much quicker than message passing over a network interface.

Poor design of the HGP architecture on a distributed system can result in frequent congestion of the network and, hence, poor performace. We illustrate an example in Figure 4.8. In this design, we have one 'master' node and 7 'slave' nodes. The HGP is constructed by the master node and the leaves of the HGP are distributed onto the slave nodes. At every evaluation of the likelihood/prediction function, the master sends requests on to each slave node and waits for the output from the slave GPs before recombining them. This design is simple and easy to manage, since we can set up the slave nodes as job servers and simply dispatch jobs from the master node to the slave nodes. This is however, inefficient, since the master node has to handle the network communications between itself and all the slave nodes. This design is an inefficient allocation of resources, prone to congestion at the master node, and leaves the connections between the slave

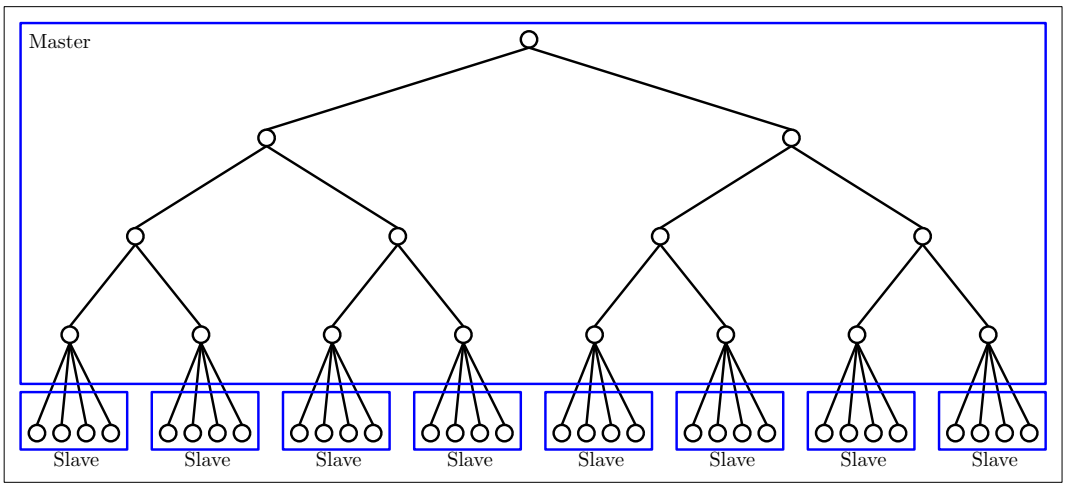

Figure 4.8: Each blue area represents the responsibility (of the computational work as part of the HGP) of each computational node in the network. The 'master' node coordinates the distribution of work to the 'slave' nodes and recombines the results for the entire HGP tree.

nodes unused. A more efficient design is illustrated in Figure 4.9. Starting from the top of the HGP tree divide the number of computational nodes available into size($\mathcal{H}$) groups, and assign each child-GP/child-HGP one group. We do this recursively until we reach the leaves of the HGP or until the size of each group reaches one. We see that this results in a more uniform distribution of network communications and computational load among all nodes.

Efficient Design

The smallest unit of computation in the HGP model is the leaf-GP. Assuming that every node in our network has the same hardware configuration, a single-threaded implementation[3] of the basic operations (mathematical functions and linear algebra routines) is used, then we would require (number of nodes $\times$ cores per node) leaf-GPs to maximise the usage of resources. As we have discussed in the Chapter 4.2.3, a deep architecture generally performs better than a wide one, thus we propose the following way to select the HGP architecture to suit the hardware available.

Let $(s_0, \cdots, s_{l-1})$ denote the sizes of the nodes at each level of a balanced HGP. leaves($\mathcal{H}$) = $\prod_{i=0}^{l-1} s_i$. As mentioned, we want leaves($\mathcal{H}$) = (number of cores). To achieve this, we simply choose $s_0, \cdots, s_{l-1}$ to be the factors of leaves($\mathcal{H}$). For example, if we have 150 nodes with 4 cores each, we would like to have $150 \times 4 = (2 \times 3 \times 5^2) \times (2^2)$ leaf-GPs. A possible architecture is $(2, 3, 5, 5, 2, 2)$.

[3]If we had a multithreaded implementation of the basic operations would, then (number of nodes) leaf-GPs would achieve maximum utilisation.

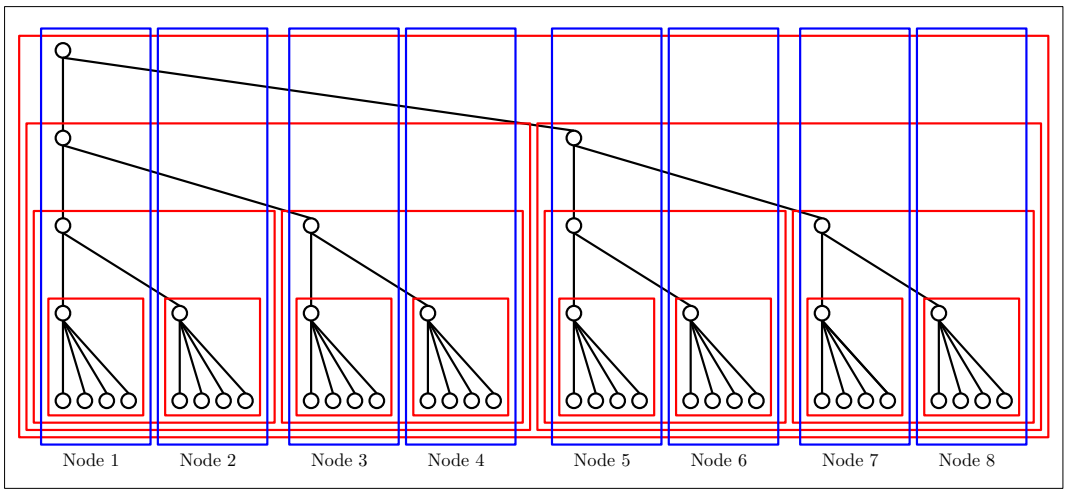

Figure 4.9: Distribution of the computational work of a HGP on a distributed system. Each blue rectangle represents one computational node, and each red rectangle represents a HGP. The overlaps between the area within the red and blue rectangles represent the computational nodes available to the HGP.

4.4 Python Implementation

Python was chosen as the language to prototype and test the HGP model in practice. Python is a expressive high level object-oriented programming language that provides simple interfaces for many of the low level primitives required for our model implementation. These include powerful numerical and scientific computation modules (NumPy/SciPy), multiprocessing, and networking modules.

4.4.1 Object Oriented Design Overview

Object oriented programming allows the abstraction of data ('attributes') into logical sets ('objects') of different types ('classes') with associated procedures ('methods') which model after real world objects and their behaviour, enabling us to develop highly complex structures with much simplicity. This is key to writing code that implements the HGP model in a logically tractable manner, since the model involves many interactions between objects such as child-GPs and parent-GPs on multiple levels, and data sets and partitions. Here we give a list of the classes implemented and briefly describe their role.

- `DataSet`

  The `DataSet` class implements the functionality for handling the data used for regression

problems. It contains a matrix of input variables, and (for training data) a vector of observed responses. This class manages the partitioning of data (using methods described previously) and allows subsets to be recombined easily. The section 'Memory Management' (Chapter 4.4.3) elaborates more about this class.

- `GP`

  An instance of the `GP` class corresponds to a GP model. Its main attributes are, a training data set, specification of the covariance function which it is equipped with, and a set of hyperparameters. Its main methods are, `train`, which trains the GP model, and `predict` which computes predictive means and variances for prediction input points.

- `HGP`

  The `HGP` class is a subclass of `GP`. It corresponds to a HGP model and implements the same interface as the `GP` class for GP modelling, therefore we can easily use HGPs as child-GPs and build up the hierarchical GP model we have introduced. In addition to the attributes inherited from `GP`, its main attributes include a list of references to its children GPs (other `GP`/`HGP` objects).

- `GPnode`*

  The `GPnode` class manages the physical resources of a computer within a network (of other computers) on which a distributed HGP is computed on. It behaves like a server and responds to request to create `GP`/`HGP` objects which are part of a larger `HGP` implemented on the network. Its main attributes are a list of references to `GP`/`HGP` objects which uses its computational resources, and information about the other nodes in the network.

- `GPmaster`*

  The `GPmaster` The `GPmaster` class manages the central node in a network of computers on which a distributed HGP is computed on. Only one instance of the `GPmaster` will be run across the whole network. On the surface, it provides the interface for the user to create and run HGPs on the network without exposing the network interface to the user, and beneath it manages the distribution of the computational workload evenly across all nodes in the network. Like a `GPnode`, its key attributes contain information relevant to the other nodes in the network, and a list of `HGP`/`GP` objects created by the user. It also has a `GPnode` object which manages the computational resources of the machine it runs on.

(* - only used for distributed computation)

4.4.2 True Concurrency in Python

Global Interpreter Lock

A known issue of the CPython interpreter (which we use in our implementation) is the lack of true concurrency using the in-built threading library. Due to the *Global Interpreter Lock* (also known as the GIL, which is implemented in the interpreter because Python's memory management is not thread safe), the greater details of which we shall not discuss here, only one single thread of Python code can be executed at any point in time. Therefore, the use of threads in the Python context only provides logical concurrency in terms of the flow of programs, but not true simultaneous computations.

Working Around the GIL Problem using Processes

Fortunately, there exists a workaround for this, via the use of processes instead of threads to perform simultaneous computations. In the POSIX [14] model, threads are lightweight units of computations belonging to the same process thus sharing the same memory space. Processes have their own memory space and come with increased system overheads compared to threads. However, on Linux (which we use for this implementation), the creation of duplicate processes ('forking') does not incur large memory costs since Linux implements a copy-on-write [15] model. This means that when a process forks into two, the memory is not copied, unless the new process attempts to modify it. In the context of our implementation, when we use additional processes to perform the computations for the child-GPs, we make no modification to the training data which is common among all child-GPs. In terms of the memory usage, each child-GP only needs to compute its own kernel matrix and the derivatives matrix which have no interaction with any other child-GP. Therefore, computing each child-GP using a separate process does not incur any large, redundant memory costs that would not be present in a true concurrency model implemented by native threads.

4.4.3 Memory Management

In our model we have overlapping subsets of data for each child-GP. If one were to add up the number of data points at the leaf-GPs of a HGP, it would exceed the number of data points in the

initial data set. Using method C3 (described in the Chapter 4.1) to combine partitions, we would have twice the number of data points at every level. This leads to $2^l N$ data points for a HGP with l levels, which grows exponentially and can easily exceed the available memory on most systems if a deep architecture is used.

This duplication of memory can be prevented, since the main training data set is shared and does not get modified. Therefore each child-GP can have access to its training data subset without copying any of it, hence avoiding the memory consumption we described in the previous paragraph. To do this, we implement a `DataSet` class, which manages the data for the GP/HGP classes. There will be *a single* instance of this class, which holds all the data in the full training set. We can create additional `DataSet` instances by invoking the `subset` method on the `DataSet` object. We specify a set of integers corresponding to the indices of the data points (which we require in the subset) in the main data set. A new `DataSet` instance is then created with no actual data, but a list of indices, and a reference to the main `DataSet` object as its superset. The only exception to this occurs when distributed computing is used. In which case, the subset of data that is required at a different machine on the network is copied from one memory space to another, and a new 'main' `DataSet` object is created.

4.4.4 Remote Object Management

Managing network communications for a distributed system poses a large challenge, since there are many details one has to manage (e.g., retrying failed message transmissions, timeouts). This adds much complexity to the system and may cause unnecessary failures in the system if not properly implemented. The *remote procedure call* (RPC) protocol enables all of the issues at the network layer to be abstracted, and allows on to use objects on a remote host with the same interface as local objects. This is done by having a dummy object on the client ('client stub'), which provides the interface of the object, but instead of executing methods on the client, invokes the required computation on a remote host ('server'), and (upon completion of the remote computation), receives the results from the remote host and returns. With this interface, the code generally remains simple and readable.

In our implementation, we use the Pyro4 [16] library, which implements remote procedural calls in Python. In addition to the `GPnode` and `GPmaster` objects which use RPC to enable communication between nodes, the classes described above can be set up as remote objects (as required). This allows the exact functionality of all classes for both local and distributed computation.

4.5 Summary

We have discussed a number of aspects of selecting an architecture for the HGP model and highlighted their strengths and likely situations in which they should be used. However, in machine learning, there is no one model that fits all problems, and our model is no exception. Therefore, we do not recommend any default choice of architecture. Instead, the architecture should be treated as a set of meta-model parameters and be chosen based on the problem at hand. The choice of architecture can be made by evaluating candidate architectures using methods such as cross-validation.

Chapter 5

Experiments

5.1 Performance

We measure[1] the amount of time required for the HGP, the full GP and the SPGP to evaluate the negative log marginal likelihood and its gradients with respect to the kernel hyperparameters. The marginal likelihood is the objective function to be maximised during GP training. Therefore, the time required for computing the marginal likelihood and its gradient is proportional to the training time and scales in $\mathcal{O}(N^3)$. The HGP was run on 64 machines[2] over a network. The architecture of the HGP was expanded as the size of the training data increased. A plot of the results is show in Figure 5.1.

The computational time required for the HGP to compute the marginal likelihood and gradients is significantly lower than that of the full GP, and we could scale it up to $2^{24} \approx 1.6 \times 10^7$ training data points. The plot demonstrates that for any problem size, we are able to find an architecture that allows us to compute the likelihood (hence train the model) within a feasible amount of time. As mentioned previously, the choice of architecture is dependent on the data, and therefore in practice, there may be less flexibility in architecture selection. However, this generally should not pose a problem since we have shown previously that as the number of data points grows, the capacity to have more child-GPs increases (Chapter 4.2).

[1] A Python implementation was used for all experiments.

[2] Each machine runs an Intel Core i7-3770 processor with 4 cores and 16GB of memory.

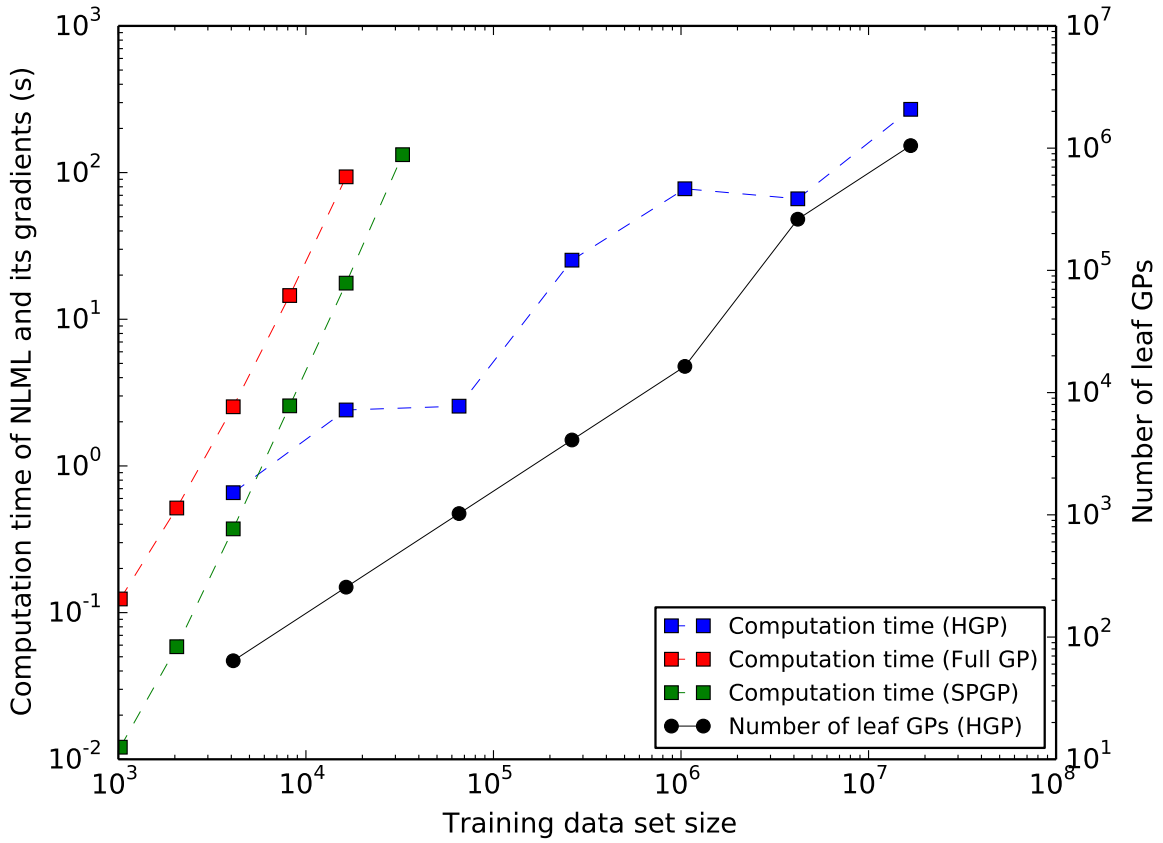

Figure 5.1: The red, green, and blue lines show the computational time required for the full GP the SPGP, and the HGP to complete one evaluation of the negative log marginal likelihood function and its gradients. The green line shows the number of leaf-GPs present in the architecture used for the HGP. All axes on log scales.

5.2 Performance on Real Data

For the Boston housing data set, the abalone data set, and the kin40k data set, we compare the HGP and full GP, presenting the standardised mean squared error[3] (SMSE) and likelihood ratio[4] of the HGP to the full GP. We also apply the HGP model to a novel bioengineering problem. In all experiments, we train the GPs/HGPs by minimising the negative log marginal likelihood using the Broyden-Fletcher-Goldfarb-Shanno (BFGS) [17] algorithm.

[3]The standardised mean squared error is given by $\frac{1}{N_*}\frac{||\mathbf{f}_*-\mathbf{y}_*||^2}{\mathrm{Var}(\mathbf{y}_*)}$ where N_* is the number of prediction points, $\mathbf{f}_*$ is the vector of predictions and $\mathbf{y}_*$ is vector the actual observed values.

[4]See Appendix A.

5.2.1 Boston Housing Data

The Boston housing data set contains information collected by the United States Census Service for census tracts in the Boston Standard Metropolitan Statistical Area in 1970. The data contains 14 variables[5] including the median home values, per capita crime rates, and average number of rooms per dwelling. This data set was used by Harrison and Rubinfeld in the 1978 paper *Hedonic Prices and the Demand for Clean Air* [18]. We predict the median home value as a function of the other 13 variables.

- Training set size: 455
- Test set size: 51
- Input dimension: 13

Model	Architecture (HGP)	Training Time (s)	BFGS Iterations	Out-of-sample SMSE	Likelihood Ratio
GP	-	26	113	0.0926	1
HGP	(3)	9.3	70	0.0889	0.989
	(3,3)	5.2	40	0.224	0.963
	(4)	8.4	92	0.163	0.977
	(4,4)	8.5	78	0.198	0.972

The single layer HGP with 3 child-GPs performed very similarly to the full GP, and training took just over a third of the time required for that of a full GP. However, the performance of HGPs with larger architectures quickly deteriorated. We believe this is due to the small size of the training data set.

5.2.2 Abalone Data

The abalone data set originates from a non-machine learning study 'The Population Biology of Abalone from the North Coast and Islands of Bass Strait' [19]. The each point in the data contains 9 measurements of an abalone, which include gender, length, diameter, height, a set of four different weights, and the number of rings. The number of rings determines the age of the abalone, and

[5]The full list of variables are: per capita crime rate by town, proportion of residential land zoned for lots over 25,000 sq.ft., proportion of non-retail business acres per town, Charles River proximity, nitric oxides concentration, average number of rooms per dwelling, proportion of owner-occupied units built prior to 1940, weighted distances to five Boston employment centres, index of accessibility to radial highways, full-value property-tax rate, pupil-teacher ratio by town, $1000(B-0.63)^2$ where B is the proportion of blacks by town, % lower status of the population, Median value of owner-occupied homes.

is obtained by cutting the abalone shell, staining it, and counting the rings. This process is a labourious task, and we try to predict the number of rings given the other measurements.

- Training set size: 3133
- Test set size: 1044
- Input dimension: 8

Model	**Architecture (HGP)**	**Training Time (s)**	**BFGS Iterations**	**Out-of-sample SMSE**	**Likelihood Ratio**
GP	-	334	25	0.449	1
HGP	(3)	411	52	0.429	0.995
	(3,3)	212	20	0.442	0.998
	(4)	111	20	0.444	0.998
	(4,4)	336	62	0.430	0.994

The HGP was able to achieve very similar results as the full GP for all the different architectures used. We note that for some architectures the HGP required a longer training time than the full GP. This was however due to the fact that the HGP took more optimisation iterations than the full GP to reach convergence.

5.2.3 kin40k

The kin40k data set is generated with a simulation of the forward dynamics of an 8 link all-revolute robot arm. The task in the data set is to predict the distance of the end-effector from a target using joint positions and twist angles [20].

- Training set size: 10000
- Test set size: 30000
- Input dimension: 8

Model	Architecture (HGP)	Training Time (s)	BFGS Iterations	Out-of-sample SMSE	Likelihood Ratio
GP	-	13330	61	0.0822	1
HGP	(4)	4688	62	0.107	0.992
	(4,4)	3505	62	0.152	0.978
	(4,4,4)	4007	77	0.215	0.956
	(4,4,4,4)	3260	66	0.335	0.909
	(4,4,4,4,4)	2223	69	0.460	0.875
	(4,4,4,4,4,4)	940	55	0.635	0.834
	(4,4,4,4,4,4,4)	1475	67	0.746	0.815

The basic single level HGP was able to achieve very similar results in a significantly shorter amount of time. Also, as we have previously seen, as we expanded the HGP architecture, the performance of the HGP decreases.

5.2.4 Hand Data

The hand data[6] set consists of 18 angle measurements from sensors on a cyberglove (see Figure 5.2) worn by a test subject to perform a certain task. We select one of the tasks for one subject and use the HGP on it.

- Training set size: 16384
- Test set size: 8194
- HGP architecture used: $(4, 4, 4, 4, 4)$ - 6 level HGP with 4 child-GPs per HGP.

Each angle measurement represents the angle of a joint at a certain point in time as the subject performs the given task. We model the angles on the middle finger (angles 7 and 8) and ring finger (angles 11 and 12) using the angles from the other parts of the hand. This simulates the problem of a synthetic finger fulfilling the actions of a missing finger, by predicting the intended action of the missing finger using the positions of the other fingers.

Missing Finger	Joint	Training Time (s)	Input Dimension	Out-of-Sample MSE (Radians)	Out-of-Sample MSE (Degrees)
Middle	7	425	16	0.174	9.96
	8	425	16	0.105	6.01
Ring	11	357	16	0.111	6.35
	12	332	16	0.099	5.67

[6]Data kindly provided by the PI Brain & Behavior Lab, Department of Bioengineering, Imperial College London.

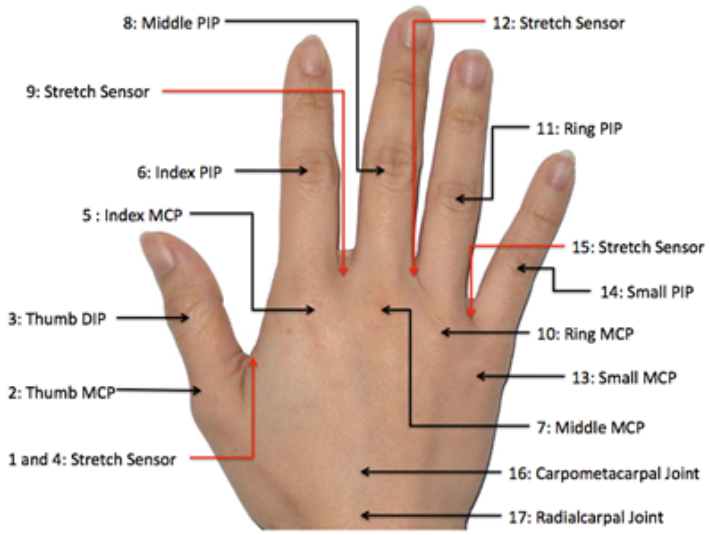

Figure 5.2: Each label depicts the position of the sensors of the cyberglove and the joint angle they measure. Figure taken from [21].

The HGP was able to achieve an out-of-sample MSE of less than 10 degrees for all four cases, allowing the actions of a real finger to be simulated by a synthetic one.

5.3 Summary

We have shown the effectiveness of the HGP on a variety of data sets and its ability to match the performance of a full GP, at a much lower computational cost. While the range of data sets tested on is not exhaustive, we believe that this serves as a proof of concept that the HGP can be useful in real-world problems.

Chapter 6

Conclusion

6.1 Summary

We have presented a conceptually simple and practical hierarchical model for applying Gaussian processes to large data set sizes ($> 10^7$), advancing current state-of-the-art Gaussian process models by at least an order of magnitude in terms of data set size. Key to the success of our HGP model is that it can be massively parallelised, i.e. the computational (and memory) load can be distributed to many computational units. We devised a computationally efficient way of analytically combining local computations in a recursive way, i.e. no sampling is required. Therefore, given sufficient computing power, the HGP can in principle handle arbitrarily large data sets. However, even with limited computing power, our HGP model can train a GP with a million data points in less than half an hour on a standard laptop, making it a practical alternative to sparse GP methods. We have demonstrated the effectiveness of our model on both synthetic and real world data sets, as well as its potential to scale massively on modern-day computer hardware.

6.2 Future Work

There exists an enormous potential for further development of the HGP model. Although the HGP only approximates a full Gaussian process, it has all the properties of a full GP (e.g. closed-form training and inference and non-degeneracy) and allows for applying the full breadth of GP research to be applied, such as heteroscedastic modelling, classification, non-Gaussian likelihoods, and even local sparsifications at the level of child GPs. In future, we will also investigate bounding the approximation error, potentially based on ideas developed in [22]. This would be an important step forward for the development of this model since it would give us an insight to the robustness

of the model, especially when applying it to unseen data. Due to limited time and resources, we were not able to perform a full training of the HGP on a large scale ($N \gg 10^7$) data set. We believe that this is an important experiment to carry out, since the main objective of the HGP is to handle data set sizes on that scale, and this would serve to validate its relevance and effectiveness in the real world.

Appendix A

Likelihood Ratio

Let $G_1 = \mathcal{N}\left(\mu_1, \sigma_1^2\right)$ and $G_2 = \mathcal{N}\left(\mu_2, \sigma_2^2\right)$ be two Gaussian distributions. We compare G_2 to G_1 by evaluating the ratio of the likelihood of G_2 to G_1 given observations drawn from G_1. We can do this empirically by drawing N independent samples $y_1, \cdots, y_N$ from G_1, and evaluate the likelihood ratio $\mathrm{LR}_{G_1}(G_2) \coloneqq \prod_{i=1}^{N} \frac{p(y_i|G_2)}{p(y_i|G_1)} = \exp\left\{-\sum_{i=1}^{N} \log\left(\frac{p(y_i|G_1)}{p(y_i|G_2)}\right)\right\}$. Here y_i are the independent observations of Y and $p(\cdot|G_2)$ is the likelihood function of G_2 (Gaussian pdf). We write the likelihood ratio as the exponential of the negative sum of log-likelihood ratios to use the Kullback-Leibler (KL) divergence to compute this in closed form, instead of drawing samples.

$$\sum_{i=1}^{N} \log p(y_i|G_j) \propto \frac{1}{N} \sum_{i=1}^{N} \log p(y_i|G_j) \overset{N\to\infty}{\approx} \mathrm{E}_{G_1}\left[\log p(Y|G_j)\right]$$

With this substitution,

$$\begin{aligned}
\sum_{i=1}^{N} \log \frac{p(y_i|G_1)}{p(y_i|G_2)} &= \sum_{i=1}^{N} \log p(y_i|G_1) - \sum_{i=1}^{N} \log p(y_i|G_2) \\
&\propto \sum_{i=1}^{N} \frac{1}{N} \log p(y_i|G_1) - \frac{1}{N} \sum_{i=1}^{N} \log p(y_i|G_2) \\
&\overset{N\to\infty}{\approx} \mathrm{E}_{G_1}\left[\log p(Y|G_1)\right] - \mathrm{E}_{G_1}\left[\log p(Y|G_2)\right] \\
&= \mathrm{E}_{G_1}\left[\log p(Y|G_1) - \log p(Y|G_2)\right] \\
&= \mathrm{E}_{G_1}\left[\log\left(\frac{p(Y|G_1)}{p(Y|G_2)}\right)\right] = D_{\mathrm{KL}}(G_1||G_2)
\end{aligned}$$

the likelihood ratio becomes $\mathrm{LR}_{G_1}(G_2) = \exp\left\{-D_{\mathrm{KL}}(G_1||G_2)\right\}$ which we can evaluate in closed form for two Gaussian distributions G_1 and G_2. Since $D_{\mathrm{KL}}(G_1||G_2) \in [0, \infty)$ and is continuous in all the parameters of G_1 and G_2, $\mathrm{LR}_{G_1}(G_2) \in (0, 1]$. This gives us a value between zero and one which we can interpret as the similarity of G_2 and G_1. Next we show that $\mathrm{LR}_{G_1}(G_2)$ is maximised globally at $(\mu_2, \sigma_2^2) = (\mu_1, \sigma_1^2)$ by showing that the KL-divergence of one Gaussian distribution from

another has a single global minimum (of 0) when both Gaussians have identical parameters. We use the following intermediate result.

$$\begin{aligned}
\mathrm{E}_{G_1}\left[\log p(Y|G_2)\right] &= \mathrm{E}_{G_1}\left[-\frac{(Y-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}\log\left(2\pi\sigma_2^2\right)\right] \\
&= -\frac{1}{2\sigma_2^2}E_{G_1}[Y^2 - 2Y\mu_2 + \mu_2^2] - \frac{1}{2}\log\left(2\pi\sigma_2^2\right) \\
&= -\frac{1}{2\sigma_2^2}\left(\sigma_1^2 + \mu_1^2 - 2\mu_1\mu_2 + \mu_2^2\right) - \frac{1}{2}\log\left(2\pi\sigma_2^2\right) \\
&= -\frac{1}{2}\left(\frac{\sigma_1}{\sigma_2}\right)^2 - \frac{(\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}\log\left(2\pi\sigma_2^2\right)
\end{aligned}$$

$$\begin{aligned}
D \coloneqq D_{\mathrm{KL}}(G1||G2) &= \mathrm{E}_{G_1}\left[\log\left(\frac{p(Y|G_1)}{p(Y|G_2)}\right)\right] \\
&= \mathrm{E}_{G_1}\left[\log p(Y|G_1)\right] - \mathrm{E}_{G_1}\left[\log p(Y|G_2)\right] \\
&= \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial D}{\partial \mu_2} &= \frac{\mu_2 - \mu_1}{\sigma_2^2} = 0 \Rightarrow \mu_2 = \mu_1 \\
\frac{\partial^2 D}{\partial \mu_2^2} &= \frac{1}{\sigma_2^2} > 0
\end{aligned}$$

$\Rightarrow \mu_2 = \mu_1$ minimises $D_{\mathrm{KL}}(G_1||G_2)$ globally for all σ_2^2.

$$\begin{aligned}
\frac{\partial D}{\partial \sigma_2} &= \frac{1}{\sigma_2} - \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{\sigma_2^3} = 0 \Rightarrow \sigma_2 = \sqrt{\sigma_1^2 + (\mu_1-\mu_2)^2} = A \\
\frac{\partial^2 D}{\partial \sigma_2^2} &= -\frac{1}{\sigma_2^2} + 3\frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{\sigma_2^4} = -\frac{1}{\sigma_2^2} + 3\frac{A^2}{\sigma_2^4} \\
&= \frac{1}{\sigma_2^4}(-\sigma_2^2 + 3A^2) = 2/A^2 \text{ when } \sigma_2 = A \\
&= 2(\sigma_1^2 + (\mu_1-\mu_2)^2)^{-1} > 0
\end{aligned}$$

$\Rightarrow \sigma_2 = \sqrt{\sigma_1^2 + (\mu_1-\mu_2)^2}$ minimises $D_{\mathrm{KL}}(G_1||G_2)$ globally for all μ_2^2

This gives us a single global minimum for $D_{\mathrm{KL}}(G_1||G_2) = 0$ at $(\mu_2, \sigma_2) = (\mu_1, \sigma_1)$ (when both distributions are identical). The unique minimum of also shows that $D_{\mathrm{KL}}(G_1||G_2)$ is unbounded.

Therefore $D_{\mathrm{KL}}(G_1||G_2) \in [0, \infty) \Rightarrow \mathrm{LR}_{G_1}(G_2) = \exp\left\{-D_{\mathrm{KL}}(G_1||G_2)\right\} \in (0, 1]$ The likelihood ratio $\mathrm{LR}_{G_1}(G_2)$ is a monotonic decreasing function of $D_{\mathrm{KL}}(G_1||G_2)$ and is not symmetric between (the parameters of) G_1 and G_2. In the comparisons we make, we set G_1 to be the predictive distribution of the full GP, which we assume is 'correct' and G_2 to be the predictive distribution of the HGP. $\mathrm{LR}_{G_1}(G_2)$ then tells us how well the HGP models after the full GP.

Bibliography

[1] C. E. Rasmussen and C. K. I. Williams. "Gaussian Processes For Regression". In: *Advances in Neural Information Processing Systems 8* (1996), pp. 514–520.

[2] Christopher K.I. Williams and Matthias Seeger. "Using the Nyström Method to Speed up Kernel Machines". In: *Advances in Neural Information Processing Systems*. 2001, pp. 682–688.

[3] Joaquin Quiñonero-Candela and Carl E. Rasmussen. "A Unifying View of Sparse Approximate Gaussian Process Regression". In: *Journal of Machine Learning Research* 6.2 (2005), pp. 1939–1960. URL: http://jmlr.csail.mit.edu/papers/volume6/quinonero-candela05a/quinonero-candela05a.pdf.

[4] James Hensman, Nicolò Fusi, and Neil D. Lawrence. "Gaussian Processes for Big Data". In: *Proceedings of the Conference on Uncertainty in Artificial Intelligence*. Ed. by A. Nicholson and P. Smyth. AUAI Press, 2013. URL: http://auai.org/uai2013/prints/papers/244.pdf.

[5] Michalis K. Titsias. "Variational Learning of Inducing Variables in Sparse Gaussian Processes". In: *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*. 2009.

[6] Miguel Lázaro-Gredilla et al. "Sparse Spectrum Gaussian Process Regression". In: *Journal of Machine Learning Research* 11 (2010), pp. 1865–1881. URL: http://jmlr.csail.mit.edu/papers/v11/lazaro-gredilla10a.html.

[7] Yirong Shen, Andrew Y. Ng, and Matthias Seeger. "Fast Gaussian Process Regression Using KD-Trees". In: *Advances in Neural Information Processing Systems*. 2006.

[8] J. Quinonero-Candela and C. E. Rasmussen. "A Unifying View of Spares Approximate Gaussian Process Regression". In: *Journal of Machine Learning Research 6* (2005), pp. 1939–1959.

[9] Z. Ghahramani and E. Snelson. "Sparse Pseudo-inputs Gaussian Processes". In: *Advances in Neural Information Processing Systems 18* (2005).

[10] Robert A. Jacobs et al. "Adaptive Mixtures of Local Experts". In: *Neural Computation* 3 (1991), pp. 79–87. URL: http://www.cs.toronto.edu/~hinton/absps/jjnh91.pdf.

[11] Edward Meeds and Simon Osindero. "An Alternative Infinite Mixture of Gaussian Process Experts". In: *Advances in Neural Information Processing Systems*. Vol. 18. The MIT Press, 2006, p. 883.

[12] Carl Edward Rasmussen and Zoubin Ghahramani. "Infinite Mixtures of Gaussian Process Experts". In: *Advances in Neural Information Processing Systems*. Vol. 2. The MIT Press, 2002, pp. 881–888.

[13] Chao Yuan and Claus Neubauer. "Variational Mixture of Gaussian Process Experts". In: *Advances in Neural Information Processing Systems*. 2009, pp. 1897–1904.

[14] Daniel Robbins. *POSIX Threads Explained*. URL: https://www.ibm.com/developerworks/library/l-posix1/.

[15] *Linux Memory Management Overview*. URL: http://www.tldp.org/LDP/khg/HyperNews/get/memory/linuxmm.html.

[16] Irmen de Jong. *Python Remote Objects*. URL: https://pythonhosted.org/Pyro4/.

[17] *Broyden-Fletcher-Goldfarb-Shanno algorithm*. URL: http://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno_algorithm.

[18] D. Harrison and D. L. Rubinfeld. "Hedonic Prices and the Demand for Clean Air". In: *Journal of Environmental Econimics and Management* 5 (1978), pp. 81–102.

[19] S. R. Talbot A. J. Cawthorn W. J. Nash T. L. Sellers and W. B. Ford. "The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait". In: *Sea Fisheries Division, Technical Report No. 48* (1994).

[20] *Kin Family of Datasets*. URL: http://www.cs.toronto.edu/~delve/data/kin/desc.html.

[21] Jovana Belic. "Classification and Reconstruction of Manipulative Hand Movements Using the CyberGlove".

[22] Si Si, Cho-Jui Hsieh, and Inderjit Dhillon. "Memory Efficient Kernel Approximation". In: *Proceedings of the 31st International Conference on Machine Learning*. 2014, pp. 701–709.