Extending Deep GPs: Novel Variational Inference Schemes and a GPU Implementation

Maximilian Ekanem Chamberlin

Department of Engineering

M.Phil in Machine Learning, Speech and Language Technology

This dissertation is submitted for the degree of

Master of Philosophy

Clare College August 2016

Declaration

I, Maximilian Ekanem Chamberlin of Clare College, being a candidate for the M.Phil in Machine Learning, Speech and Language Technology, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Word count – total word count excluding appendices:

10,189

Signed: [illegible]

Date: 12 AUG 16

Maximilian Ekanem Chamberlin
August 2016

Acknowledgements

I would like to thank my supervisor Rich for the kind support he has offered as a mentor during this project. I would also like to thank the many great personalities on my course who made studying here, at Cambridge, a real pleasure. I want to thank my parents Gary and Marie for their limitless support and love. And last but not least, I want to thank Sophie for the energy, the encouragement and for putting up with it all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction: The Deep Gaussian Process Model

1.1 What are Deep GPs?

A Deep Gaussian Process (DGP) is a model that arises from the composition of Gaussian Processes. Each layer of a DGP may be regarded as a GP mapping with noise, where the outputs of one layer become the inputs to the following layer. A formal definition of a DGP is given below:

1.1.1 DGP Definition:

A one dimensional DGP models outputs y given inputs x according to:

$$p(y|x) = \int_h p(y|h_{L-1}) \left[\prod_{l=2:L-1} p(h_l|h_{l-1}) \right] p(h_1|x)$$

We model the conditional outputs at each layer as draws from a GP with a noise, i.e:

$$p(h_l|h_{l-1}) = \int_{f_l} p(h_l|f_l) p(f_l|h_{l-1})$$

Where:

$$p(h_l|f_l) = N(h_l; f_l, \sigma_l^2 I)$$

$$p(f_l|h_{l-1}) = N(f_l; 0, K_{nn}^l)$$

Here K_{nn}^l is the kernel evaluated at input h_{l-1}, with the subscript n denoting the size $(n \times n)$. Implicitly, we may also rewrite $p(h_l|h_{l-1})$ by marginalising out the GP function draws f_l, so:

$$p(h_l|h_{l-1})=GP(0, K^l_{nn}+\sigma_l^2 I)$$

1.1.2 Sampling from a DGP

Knowing how to sample from a model often provides a more concrete understanding of how the model works than a first sight of the formal definition. Consider the graphical model in fig. 1.1, which describes the generative procedure for the outputs of a DGP, given inputs .

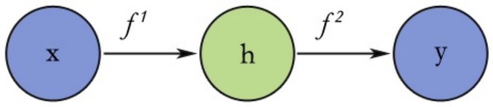


Figure 1.1: A Deep Gaussian Process

Given our input data x, we first sample a function f^1 over the domain x, according to the first layer's GP: $f^1_{|x} \sim N(0, K^1_{nn})$. The reason $f^1_{|x}$ is drawn from a distribution with zero mean is that it simplifies inference, but does not greatly restrict our modelling capacity.

Having done this, to obtain samples of h, we simply add noise to the function draw $f^1_{|x}$. Equivalently, we may simply sample h from $N(0, K^1_{xx}+\sigma_1^2 I)$ instead.

After sampling h, we follow a similar procedure, taking samples y according to the GP at the second layer with h now considered as the input, so $y \sim N(0, K^2_{hh}+\sigma_2^2 I)$. To extend this sampling procedure to a GP with more hidden layers, at each intermediate layer we must also sample h_l given the samples from the previous layer, h_{l-1}.

For multi-dimensional data, the sampling procedure is again very similar. The only difference being that, at each layer, we have multiple GPs - one for each output dimension at that layer. Please see [Damianou 2015] for further description.

Sampling from a DGP is relatively straightforward, however inference is much more challenging and is not analytically tractable. Approximate inference schemes- in particular those based on variational free energy- may be used to perform inference tractably. This will be a subject of exploration in later sections of this thesis (Chapters 2 and 4).

1.2 Why Deep Gaussian Processes?

DGPs are an interesting class of models for both researchers and machine learning practitioners alike. For practitioners:

- DGPs have achieved state of the art results on a range of regression tasks when trained using approximate expectation propagation [Bui, Turner, Li et al 2016].
- Hensman et al. [2014] have also trained DGPs on large scale data-sets using stochastic variational inference, demonstrating the applicability to large and complex data sets.
- Parameter tuning of DGPs may be performed in a systematic manner by optimising approximations to the log-likelihood, which absolves the need for cross-validation [Duvenaud 2012].

For researchers:

- The Bayesian framework used in DGPs leads to an automatic Occam's razor that penalises complex models and guards against over-fitting [Damianou 2013; MacKay 1995].
- Damianou [2013] further demonstrates the robustness of DGP models to scarcity in small data-sets, where the risk of over-fitting is greater.
- Variational Approximations give us a principled manner of performing inference in DGPs. However, there is much that can be done in coming years to improve upon the variational approximations used in practice.

However, DGPs are a relatively recent addition to the class of models investigated by Machine Learning researchers, and one might plausibly ask why the use of a DGP is to be preferred over a standard Gaussian Processes – especially given that GPs boast many of the advantages attributed to DGPs (e.g. the Bayesian framework, robustness to small data-sets etc.). And added to this, inference is analytically tractable in a GP.

The answer to the question 'Why DGPs and not GPs?' primarily comes down to representational power. A DGP is able to model a more complex probability space over functions than a standard Gaussian Process. The joint-normal assumption of a GP's outputs (and consequently its derivatives)[1] severely impairs its ability to even model simple functions like the step function depicted below. We give an explanation for why this is so in what follows.

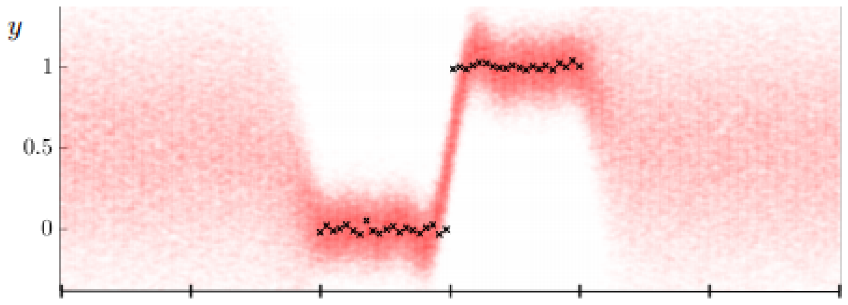


Figure 1.2: A GP attempting to model a step-function. [Source: Damianou 2015]

[1][Duvenaud 2012: For GPs with a squared exponential kernel the derivatives are point-wise distributed as $N(0, \frac{\sigma^2}{l^2})$, where σ^2 is the kernel variance and l is the kernel length-scale]

Since the derivatives of a GP are distributed normally [bell-shaped], either the most of the probability mass lies near a single mode or the distribution is relatively flat. For a step-function, the prior distribution of derivatives would at the very least need to be bi-modal and certainly not flat. This is because the distribution over function derivatives must put most of its mass at zero, some of its mass at very high derivatives, but have zero mass almost everywhere in between. For further discussion of the derivatives of a DGP, we refer the reader to [Duvenaud 2012]. However, as we shall see from the experimental work carried out in chapter 3, a DGP is not blighted by this same issue and can model a more complex probability space over functions.

More generally speaking, with the advent of neural networks, hierarchical clustering, sophisticated graphical models and many other such inference engines, it may be said that deeper architectures have begun to dominate the machine learning landscape. Deeper architectures allow for richer representations of the data to be used when performing inference or making predictions. Given this, we should not to be surprised that a DGP may model a more complicated range of functions than a Gaussian Process, since it may benefit from compositional structure, and the reuse of representations at lower levels, in much the same way as e.g. a neural network. We will see evidence of this in chapter 3 with a DGP's modelling of a sum of sinusoid.

1.3 Current Limitations of Deep GPs

In what follows, we outline a number of the limitations to modelling data with DGPs.

But first, we briefly outline the training procedures used in DGPs. These typically involve a variational approximation to the log-likelihood. Consider fig. 1.3. We have an intractable family of probabilities P (representing a family of DGP models), with each $p \in P$ a DGP model parameterised by θ. Since computing $\log p_\theta(x)$ for a data-set is hard, we use a simpler variational distribution $q_\gamma \in Q$ to form a tractable lower bound to the log-likelihood.

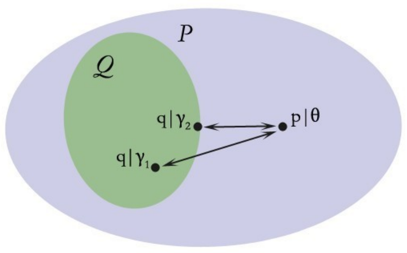


Figure 1.3: Variational Inference as KL Minimisation

We then maximise this lower bound by equivalently minimising the KL divergence $KL(q|p_\theta)$ with respect to the variational parameters γ of q. We may also simultaneously optimise the lower bound with respect to the parameters θ of p to obtain higher training-data likelihoods. So to perform inference in a DGP, we simultaneously optimise γ and θ, until we reach a good lower bound on the log-likelihood. However, such an approach is not without its difficulties.

The extra variational parameters γ that emerge from the variational framework need to be optimised in addition to the model parameters θ, which can be challenging if they are sufficiently large in number. For the standard mean-field approximations in a DGP, the number of variational parameters is of order $O(ndl)$, where n is the number of data-points, d the dimensionality of the data and l the number of layers. This is quite large, so we must either develop efficient optimisation schemes for these parameters or reduce their number. Hensman [2012], for instance, considers a natural gradient approach to efficiently optimise the variational parameters.

We may also challenge the mean-field approach to modelling q, which is fully factored over the hidden variables in a DGP. Modelling fewer dependencies results in a relatively loose lower bound on the log-likelihood p. To date, in the literature we have not seen any extensions on the mean-field approach to variational inference in a DGP.

We now move to consider how the implementation of DGPs might be improved upon. Unlike neural networks, which have benefited from a burgeoning of GPU packages to optimise deep architectures with millions of parameters, there are no such packages for variational free-energy DGPs. A GPU implementation of a deep GP in a package with GPU support like Theano, Tensorflow or Torch could greatly speed up the training of DGP models, and be of great benefit to practitioners and researchers alike who hope to scale their experimentation to large data-sets.

1.4 Questions Addressed by this Thesis

The aim of this thesis is to make good progress in tackling a number of the issues raised in the preceding section. We start with an introductory chapter that reproduces the Titsias [2009] derivations for sparse GP regression, which is the bedrock for the variational approximations used in DGPs. We do this by summarising the main points and presenting them in a way which may be more familiar to the unexposed reader.

In chapter 2, we then show how the framework developed by Titsias may be extended to DGPs, which is absent from the literature- although very similar extensions (Bayesian GPLVM) have been described. In chapter 4, we offer our most important research contribution by extending the mean-field variational inference framework in DGPs. This is done by considering a new class of variational distribution that model different kinds of dependencies, e.g. through layers, through dimension, etc..

Along the way, (in chapter 3) we will investigate Tensorflow implementations of sparse GPs, producing a careful analysis of different optimisation schemes. Having done this, we also provide a DGP implementation, based on the methods developed by Titsias 09 and Hensman [2012], who both use variational methods to perform inference in GP models. We then run tests on the model to see how well the DGP performs against competing models on some toy data-sets similar to those considered by Damianou [2015] (which includes the step-function as discussed). Ultimately, we hope that our novel DGP implementation in Tensorflow will aid researchers investigating such models, and who ultimately hope to train models over large data-sets.

In the final section of the thesis, we point to areas of research that may extend methods discussed in this thesis. In the next section, we begin by elucidating the variational inference framework used in both GPs and DGPs, with reference to Titsias 09.

Chapter 2

Variational Inference in GPs and DGPs

2.1 Executive Summary

- This section aims to explain the variational inference framework used within GPs and DGPs, thereby laying a foundation for future work. We start with a description of Titsias' 09 paper.

- Without taking a sparse approximation, Gaussian Processes are not tractable models for large data-sets. Take a function f drawn from a GP, i.e $f \sim GP(0, K)$. If we condition over a set of n training data-points x, $f|x$ is distributed normally, according to $N(0, K_{nn})$. For a sufficiently large numbers of these points, evaluating the kernel matrix K_{nn} requires $O(n^2)$ operations, and inverting it $O(n^3)$. As a result, many of the queries put to the model would become too costly.

- Sparse GP approximations of the log-likelihood replace the costly K_{nn} terms in the true log-likelihood with low rank approximations. These approximations are typically formed using the kernel matrices K_{mm} of a set of m inducing points u. Using these low rank approximations, computing the log-likelihood will cost at most $O(m^2 n)$. Regarding notation: we also write the kernel function for cross kernel, evaluated over the inputs to u and f as K_{mn}

- A very commonly used sparse approximation takes a variational free energy approach. A variational distribution over the function draws is defined: $q(f) = q(f \neq u, u) = p(f \neq u|u)q(u)$. This variational distribution ultimately decouples the outputs $f \neq u$ from one another (within the variational approximation), given a smaller set of function draws or inducing points u. The decoupling of $f \neq u$ points given inducing points u thereby eliminates the need to compute K_{nn} when forming the variational approximation to the log -likelihood. [1]

- A key question that emerges as a result of this variational free energy method is how to determine the parameters to the variational distribution $q(f) = p(f \neq u|u)q(u)$. We now need to determine the means and covariances of $q(u)$ as well as the input locations to the

[1] [**N.B.** Here we take u to be the function f evaluated at input locations z, and $f \neq u$ to mean the function f restricted to points $Dom(f)/z$].

inducing points u. Within this chapter, we will investigate two methods for determining these parameters, following Titsias [2009] who derives the optimal parameters for the variational distribution over inducing points $q(u)$ and very briefly Hensman [2013] who optimises $q(u)$ using natural gradients.

- Having explicated the approaches of both Titsias [2009] and Hensman [2013], we give a summary of how these competing methods may be put to use within a deep GP setting, a description which so far has been absent from the literature although similar cases have been touched upon [Gal et al. Bayesian GPLVM 2014]. Further, we discuss how it might be possible to make the approaches of Titsias and Hensman complementary.

2.2 Titsias' Sparse GP derivation:

Our aim will be to optimise a lower bound of the log probability of our model $\log[p(y|x)]$ with respect to the model parameters. Our model takes the form of a standard GP with noise, i.e:

$$y|f \sim N(y; f, \sigma^2\mathrm{I})$$

$$f|x \sim N(f; 0, K_{nn})$$

We could simply write down the log probability of our model directly $\log p(y|x) = \log N(y|0, K_{nn} + \sigma^2\mathrm{I})$, but this would entail a costly inversion of K_{nn} which we hope to avoid.

Instead, our aim is to find a lower bound for the log likelihood of our model using variational inference, which results in a computationally tractable bound of the log-likelihood.

We may write the log-likelihood of $y|x$, marginalising across function draws, as:

$$\log[p(y|x)] = \log \int_f p(y|f)p(f|x)$$

At this step, we rewrite the expression above, by partitioning the function f according to $f = \{f \neq u, u\}$. Here u denotes the subset of the function outputs f, evaluated at inputs z (which is a variational parameter we optimise).And, $f \neq u$ denotes the the function f, evaluated over all points except z. Then the likelihood term becomes:

$$\log[p(y|x)] = log \int_{f\neq u,u} p(y|f \neq u, x)p(f \neq u|u)p(u)$$

In the figure below, we present the graphical model with the inducing points. Note that $p(f \neq u|u)$ is a conditional GP.

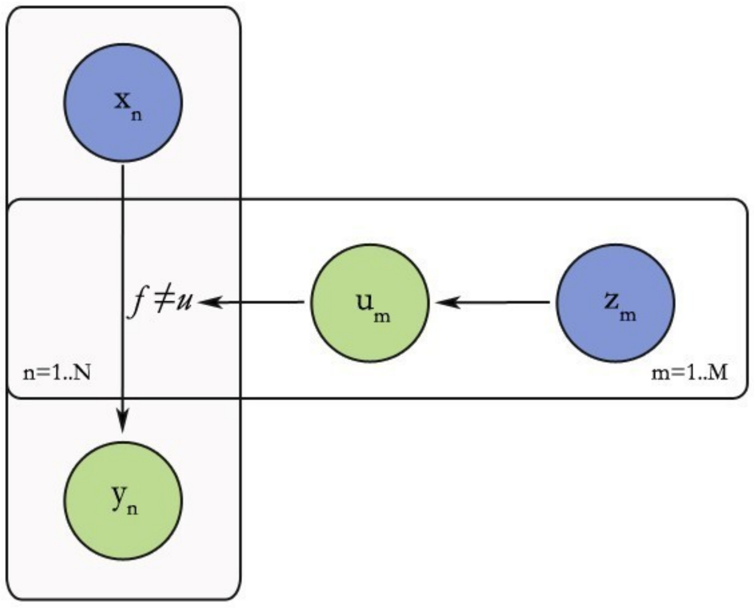


Figure 2.1: Sparse GP Model

We may now approximate this likelihood term by constructing a variational distribution over hidden variables. It will take the the form:

$$q(f \neq u, u) = p(f \neq u|u)q(u)$$

where $q(u)$ is taken to be Gaussian. So what do these variational modelling assumptions mean? Ultimately, they will enable us to write down the variational approximation so that it is factored by n. From this perspective, the approximation may be thought of as decoupling outputs $f \neq u$ in the variational lower bound. This may seem somewhat counter-intuitive at first, given that the variational distribution over $f \neq u|u$ is defined as:

$$q(f \neq u|u) = p(f \neq u|u) = N(0; K_{nm}K_{mm}^{-1}u; K_{nn} - K_{nm}K_{mm}^{-1}K_{mn})$$

And is therefore clearly jointly Gaussian with a K_{nn} term in the covariance. However, as we will see, though clever cancellations, we will be able to obviate the need to model $f \neq u$ jointly. If the reader is looking for an explanation as to why this must be so, it is to do with the complex patterns of interaction between modeling assumptions given by the variational distribution and the actual model. As a result of all this, we will be able to write down the variational approximation without calculating the full K_{nn} kernel matrix.
Using Jensen's inequality, we form the following variational lower bound:

$$\log[p(y|x)] \geq \left\langle \log \frac{p(y|f \neq u, x)p(f \neq u|u)p(u)}{q(f \neq u, u)} \right\rangle_{q(f)}$$

Which, expanding $q(f \neq u, u)$, evaluates to:

$$= \left\langle \log \frac{p(y|f \neq u)\cancel{p(f \neq u|u)}p(u)}{\cancel{p(f \neq u|u)}q(u)} \right\rangle_{q(f)}$$

As mentioned, this cancellation absolves us of the need to model $f \neq u$ jointly . For notational convenience, we now write: $\mathcal{E}(y|x,u) = \langle log[p(y|f \neq u, x)] \rangle_{p(f \neq u|u)}$, where $\mathcal{E}(y|x,u)$ is the expression for the log of a conditional GP with added noise. By rearranging the expression above, we may write the variational lower bound succinctly as:

$$F(q) = \left\langle \mathcal{E}(y|x,u) - log\frac{q(u)}{p(u)} \right\rangle_{q(u)} \tag{2.1}$$

The underlying variable in the expectation of $\mathcal{E}(y|x,u)$, $\log p(y|f \neq u, x)$, is factored by n. And so all terms in the lower bound will be factored in n. Thus, we need never compute K_{nn}.

2.3 Determining the Optimal $q(u)$

So far we have written down our variational lower bound $F(q)$ as an expectation with respect to $q(u)$ in formula 2.1. We could therefore optimsie the lower bound with respect to parameters for the distribution of $q(u)$ using gradient descent as Hensman and others do. However, according to Titsias , we may also derive an optimal distribution $q(u)$ given inputs z, which tightens the lower bound with respect to $q(u)$.

The derivation of the optimal $q(u)$ requires techniques from Lagrangian optimisation and variational calculus. The Lagrangian constraint comes from the fact that all probabilities must sum to one:

$$\int q(u) = 1$$

To solve this problem, little knowledge of variational calculus is required, given the following intuitive framework: We may think of functions as being infinitely long vectors of variables, indexed by elements of their domain. Thus, $f(u)$ is the 'u^{th}' element of the vector of variables in f. Crucially, each of the variables in $[f(u)|u \in Dom(f)]$ must be thought of as being independent from one another. So $f(u_1)$ does not affect $f(u_2)$, where $u_1 \neq u_2$, unless otherwise stated. Within this setting, we may think of integration over functions as summations over the variables in $[f(u)|u \in Dom(f)]$. Taking piecemeal into consideration, we may write down many familiar results from variational calculus.

Variational Calculus Results:

1. $\frac{\delta}{\delta f(u)} log(f(u)) = \frac{1}{f(u)}$
2. $\frac{\delta}{\delta f(u)} f(u) = 1$
3. $\frac{\delta}{\delta f(u)} \int_x f(x) = 1$

Figure 2.2: Results from Variational Calculus

Acknowledging results from variational calculus, we may differentiate the following Lagrangian to find the optimal $q(u)$:

$$\frac{\delta}{\delta q(u)}\left[F(q)-\lambda(\int q(u)-1)\right]=$$

$$\langle\mathcal{E}(y|x,u)\rangle+\log q(u)+\log p(u)+1-\lambda=0$$

Setting this expression to zero, and solving for $q(u)$ we obtain:

$$q(u)\propto e^{\langle\mathcal{E}(y|x,u)\rangle}p(u)\ (*)$$

The term $\mathcal{E}(y|x,u)=\langle log[p(y|f\neq u,x)\rangle_{p(f\neq u|u)}$ will appear in subsequent results throughout this thesis, including layer by layer bounds for a DGP. This terms is evaluated using standard results from liner algebra and is presented in the appendix. It evaluates to:

$$\mathcal{E}(y|x,u)=logN(y;K_{nm}K_{mm}^{-1}u,\sigma^2I)-\frac{1}{2\sigma^2}\mathrm{Tr}[K_{nn}-K_{nm}K_{mm}^{-1}K_{mn}] \tag{2.2}$$

2.3.1 Determining the optimal $q(u)$ distribution:

Given that we have unpacked the term $\mathcal{E}(y|x,u)$ and given our results (*) from variational calculus we may determine$q(u)$ by unpacking (*):

$$q(u)\propto e^{\langle\mathcal{E}(y|x,u)\rangle}p(u)\ (*)$$

By inspection we see that $q(u)$ is Gaussian since the highest order term in any exponents in (*) is quadratic in u. Let $\alpha_{nm}=K_{nm}K_{mm}^{-1}$. Then by unpacking terms, we find:

$$\log\left[e^{\langle\mathcal{E}(y|x,u)\rangle}p(u)\right]\propto-\frac{1}{2\sigma^2}[-2y^T\alpha_{nm}u+u^T\alpha_{nm}^T\alpha_{nm}u]-\frac{1}{2}u^TK_{mm}^{-1}u \tag{2.3}$$

As a result of this, $q(u)$ is distributed with:

$$\Sigma^{-1}=\frac{1}{\sigma^2}\alpha_{nm}^T\alpha_{nm}+K_{mm}^{-1}$$

$$\mu=\Sigma[\frac{1}{\sigma^2}(\alpha_{nm}^Ty)]$$

Given this, we may may simply plug-in the variational parameters for the optimal distribution $q(u)$.

2.3.2 Reversing Jensen's inequality to evaluate the optimal $F(q)$

Rather than plugging the parameters for $q(u)$ into the variational lower bound to determine the optimal bound, Titsias uses an elegant trick that involves reversing Jensen's inequality. However, as we will not make use of the trick in later sections of the thesis, we relegate the description of it to the appendix. Given prior calculations, the optimal bound may be written as:

$$F(q) = logN(y; 0, \sigma^2 I + Q_{nn}) - \frac{1}{2\sigma^2} TR[K_{nn} - Q_{nn}]$$

Where $Q_{nn} = K_{nm} K_{mm}^{-1} K_{mn}$

At this point, it would be appropriate to step back to make a number of remarks. The derived bound has no K_{nn} terms, which is as desired since we hoped to reduce computations to a factor $O(n)$.

However, evaluating the log likelihood term in $F(q)$ requires us to compute both the precision of $\sigma^2 I + Q_{nn}$ and the determinant of $\sigma^2 I + Q_{nn}$, which may be the cause for some concern since both are $n \times n$ matrices. In fact, this is not a problem. Using the matrix inversion lemma and the fact that Q_{nn} is a rank m matrix, we may evaluate the determinant and inverse in $O(nm^2)$.

The bound may finally be optimised with respect to the parameters of the kernel function K and the input locations z of the inducing points u. Tensorflow and Theano are packages that are well equipped to optimise these gradients.

Given this, we have obtained a variational bound that ensures training Gaussian Processes over large data-sets is tractable. We hope that we may have made a small contribution to the [Titsias 2009] paper, by stressing the intuition and providing a lighter introduction to the reader. With this frame-work in place, we will be able to extend the literature, deriving new variational approximation bounds for DGPs.

2.4 Inducing Points make Inference in DGPs tractable

Before we dive into the derivation u: not only do they make inference tractable for large data-sets in ordinary GPs, but they also make inference in a DGP tractable.

To perform inference in DGPs without pseudo-points, we'd need to be able to compute the expectation a kernel function $\langle K_{nn}^{-1} \rangle_{q(x)}$, which pipes a Gaussian $q(x)$ through a highly non-linear kernel function $\langle K_{nn}^{-1} \rangle$ which is intractable.

As has been demonstrated in the preceding section, inducing points obviate the need to compute matrices like K_{nn}^{-1} , replacing these terms kernels linear in m, like $\langle K_{nm} K_{mm}^{-1} \rangle_{q(x)}$, which are analytically tractable to compute. Because of this, inference is made tractable in a DGP. A fuller explanation of why this is so may be found in the appendix. Having mentioned this point we may proceed to analyse mean-field inference in a DGP.

2.5 Mean Field Variational Inference in a DGP

We will now present a very brief summary of mean-field variational inference in a DGP, which may be of benefit to the reader in later sections (chapter 4) where we explore new variational approximations. We may express the model's log-likelihood as:

$$p(y|x) = \int_{h,f} \prod_{l,d} \log p(h^{ld}|f^{ld} \neq u^{ld}, h^{l-1}) p(f^{ld} \neq u^{ld}|u^{ld}) p(u^{ld}) = \int_h \prod_{l,d} \mathcal{E}^{ld} p(u^{ld})$$

Here, we assume that the index l is from $1 : L$, that $y = h^L, x = h^0$. Here, just as in the previous sections, we take:

$$\mathcal{E}^{ld} = \int_f \log p(h^{ld}|f^{ld} \neq u^{ld}, h^{l-1}) p(f^{ld} \neq u^{ld}|u^{ld})$$

$$= \langle \log p(h^{ld}|f^{ld} \neq u^{ld}, h^{l-1}) \rangle_{p(f^{ld} \neq u^{ld}|u^{ld})}$$

We will now use a mean-field (over h and u) variational approximation to lower bound the likelihood. This variational distribution will take the form:

$$q(h, f) = \prod_{l,d} p(f^{ld} \neq u^{ld}|u^{ld}) q(u^{ld}) q(h^{ld})$$

In this setting $q(h^{ld})$ is modelled as a Gaussian with diagonal covariance of size n. Given this, we may write a variational lower bound for the log likelihood of the model:

$$F(q) = \sum_{l,d} \langle \mathcal{E}^{ld} \rangle_{q(h,u)} - \sum_{ld} \mathrm{KL}(q(u^{ld})|p(u^{ld})) + \sum_{l,d} \mathrm{H}(q(h^{ld}))$$

As one can see, this bound factors in l, d, so for ease of notation we drop these subscripts, and pull out the $q(u)$ term in$\langle \mathcal{E}^{ld} \rangle_{q(h,u)}$. We denote h^{l-1} as h^-.

$$F(q) = \langle \langle \mathcal{E} \rangle_{q(h,h^-|u)} \rangle_{q(u)} - \mathrm{KL}(q(u)|p(u) + \mathrm{H}(q(h))$$

Excepting the entropy term which does not depend on $q(u)$, this is almost in exactly the same form as the Titsias bound. This means that, having taken the derivatives using variational calculus, the optimal $q(u)$ will take the form:

$$q(u) \propto e^{\langle \mathcal{E} \rangle_{q(h)}} p(u) \tag{2.4}$$

We have already determined $\mathcal{E}$ from the previous section, and we need only substitute $y = h$ into the equation A.3. Given this, and ignoring terms that don't depend on u, we know that:

$$\mathcal{E} = -\frac{1}{2\sigma^2}[-2h^T \alpha_{nm} u + u^T \alpha_{mn} \alpha_{nm} u] + constant$$

Taking expectations with respect to $q(h, h^-)$, yields:

$$\langle \mathcal{E} \rangle_{q(h,h^-)} = -\frac{1}{2\sigma^2}[-2\mu_h^T \langle \alpha_{nm} \rangle u_{q(h^-)} + u^T \langle \alpha_{mn} \alpha_{nm} \rangle_{q(h^-)} u] + constant \tag{2.5}$$

This is almost exactly the same form as found in the preceding section, the only difference being that we now take expectations over the variables of interest. As a result of this, we may immediately write down the distribution for $q(u)$ as:

$$\Sigma^{-1} = \frac{1}{\sigma^2}\langle \alpha_{mn}\alpha_{nm}\rangle + K_{mm}^{-1}$$

$$\mu = \Sigma[\frac{1}{\sigma^2}(\langle \alpha_{mn}\rangle \mu_h)]$$

2.5.1 Determining the bound

To determine the bound, it is necessary to compute$\langle \mathcal{E}\rangle_{q(h)}$. The first helpful thing to note is that $\langle \mathcal{E}\rangle_{q(h)}$ is very closely related to $q(u)$. In fact, given equations A.3 and 2.5 from the previous sections, $\langle \mathcal{E}\rangle_{q(h)}$ be expressed as:

$$\langle \mathcal{E}\rangle_{q(h,h^-)} = \log N(0;0,\sigma^2 I) - \frac{1}{2}u^T(\Sigma^{-1} - K_{mm}^{-1})u + \frac{1}{\sigma^2}\mu_h^T\langle \alpha_{nm}\rangle u$$

$$-\frac{1}{2\sigma^2}\mathrm{Tr}[\Sigma_h + \mu_h^T\mu_h + \langle K_{nn} - K_{nm}K_{mm}^{-1}K_{mn}\rangle]$$

To finally determine $\langle \mathcal{E}\rangle_{q(h,h^-,u)}$, we apply the trace trick to u in the above expectation, giving us:

$$\langle \mathcal{E}\rangle_{q(h,h^-,u)} = -\frac{1}{2}Tr[(\Sigma^{-1} - K_{mm}^{-1})(\Sigma + \mu^T\mu)] + \frac{1}{\sigma^2}\mu_h^T\langle \alpha_{nm}\rangle \mu$$

$$-\frac{1}{2\sigma^2}\mathrm{Tr}[\Sigma_h + \mu_h^T\mu_h + \langle K_{nn} - K_{nm}K_{mm}^{-1}K_{mn}\rangle] + \log N(0;0,\sigma^2 I)$$

We are now in a position to determine the bound. Factorised by l, d, the bound is given by:

$$F(q) = \langle \mathcal{E}\rangle_{q(h,h^-,u)} - \mathrm{KL}(q(u)|p(u)) + \mathrm{H}(q(h))$$

By this point, all terms may be evaluated:

- $\langle \mathcal{E}\rangle_{q(h,h^-,u)}$ as given above
- $\mathrm{H}(q(h)) = \mathrm{H}(N(\mu_h, \Sigma_h))$
- KL(q(u)|p(u))=KL($N(\mu, \Sigma)|N(0, K_{mm})$)

where Σ_h is diagonal.

The cost (computational complexity) of computing the bound above for a single layer and dimension, is no different than that of a GP. It may be computed in $O(m^2 n)$, which gives an overal computational complexity of $O(nm^2 ld)$.

2.5.2 Future Directions

We have spent a good deal of time summarising the introductory material in these past few pagers, but this is not without good reason. The material provides a framework that will be expanded upon in much more detail in chapter 4, where we evaluate the variational lower bounds arising from a number of different variational approximations. These will go beyond the standard mean-field variational inference scheme, modelling dependencies between dimension, layers etc.

2.6 Natural Gradient Optimisation of Natural Parameters

In the preceding sections, we explored variational lower bounds for GPs, that make use of the optimal distribution for $q(u)$. Hensman et al [2013] takes a different approach. They consider a scheme based on optimising the parameters for $q(u)$ using natural gradients, and then evaluate the variational bound at this distribution $q(u)$.

The underpinning principle to the natural gradient approach is to take gradients not with respect to normal Euclidean distance, but with respect to a metric that better respects the 'distance' between probability distributions, for instance the symmetric KL [Honkela 07]. We do this because using the Euclidean gradient may lead to poor optimisation in certain scenarios. For instance, if we optimise a distribution q which began to have very small variances, changing the mean component of distribution q_t by even a small amount might well lead to a vastly different distribution q_{t+1}. This may ultimately lead to erratic behaviour during optimisation.

To take natural gradients of the log-likelihood L of a GP model with respect to $q(u) = N(u; m, S)$ we must first define the following:

$$\begin{gathered}\theta_1 = S^{-1}m \text{ (natural mean of q)} \\ \theta_2 = -\tfrac{1}{2}S^{-1} \text{ (precision of q)} \\ \eta_1 = m \\ \eta_2 = mm^T + S\end{gathered}$$

Since $G(\theta)$, the Fisher information matrix, is the Identity the natural gradient is given by:

$$\tilde{g}(\theta) = G(\theta)^{-1}\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \eta}$$

Given this, Hensman derives the following update equations for the natural parameters of a sparse GP :

$$\begin{gathered}\theta_{2(t+1)} = -\tfrac{1}{2}S^{-1}_{(t+1)} \\ \theta_{2(t+1)} = -\tfrac{1}{2}S^{-1}_{(t)} + l(-\tfrac{1}{2}\Lambda + \tfrac{1}{2}S^{-1}_{(t)}) \\ \theta_{1(t+1)} = -\tfrac{1}{2}S^{-1}_{(t+1)}m_{(t+1)} \\ \theta_{1(t+1)} = -\tfrac{1}{2}S_{(t)}m_{(t)} + l(\tfrac{1}{\sigma^2}K^{-1}_{mm}K_{mn}y - S^{-1}_{(t)}m_{(t)})\end{gathered}$$

Where:

$$\Lambda = \frac{1}{\sigma^2}K^{-1}_{mm}K_{mn}K_{nm}K^{-1}_{mm} + K^{-1}_{mm}$$

In chapter 3 we implement these equations as a procedure to perform the updates of our sparse GP model, which may then be compared with the work of Titsas 09.

We have in this section presented Hensman's method and Titsias' method as taking orthogonal modelling decisions. This need not be so, for we may optimally integrate out $q(u)$ as in Titsias' method but then apply natural gradient parameter updates to the distribution over $q(h)$ (in a DGP). Given the mean-field variational approximation used in a DGP, the distribution for $q(h)$ is the product of univariate Gaussians. This considerably simplifies the natural parameter gradient updates, meaning we need only scale the Euclidean gradients for the mean and variance by the inverse standard deviation, and half the inverse variance respectively. [Honkela 10a Equation 10, 11].

2.7 Concluding Remarks

Within this section we have laid bare the variational framework at the heart of both GPs and DGPs, and discussed two approaches for optimising variational parameters as given by Hensman and Titsias respectively. The framework handed down by Titsias will be used to develop theoretical results contained within chapter 4. In the next section we will move to a practical investigation of the models described, with the aim of trying to find which optimisation methods work better in practice.

Chapter 3

Testing GPs and DGPs

This chapter will discuss my implementation of sparse GPs and DGPs, as well as the experiments undertaken. We will investigate the performance of the DGP and GP models on data from a step function, replicating experimentation undertaken by Damianou [2015], as well as on data from a sum of sinusoids function.

3.1 Sparse GP Results

My implementation of a sparse GP is based on the Titsias 09 paper. For this implementation, the lower bound to the log-likelihood is evaluated at the optimal $q(u)$. Code is listed in the Appendix.

3.1.1 Sinusoidal Function

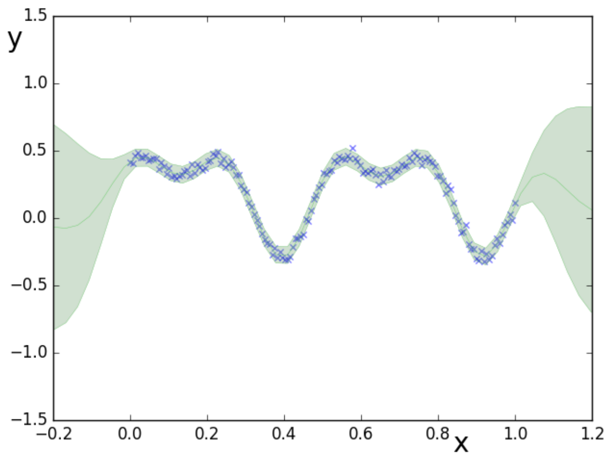


Figure 3.1: Sparse GP Modelling a Sum of Sinusoids

As can be seen from the figure, the model has little trouble modelling the curve near the training data; the variance predictably expands as we move further away from these points. The model shows a good likelihood over the test-data (Table 3.1, comparable with the DGP), and also a low mean squared error approximately 0.1 per data-point. In addition, for sufficiently many pseudo-points, the sparse GP model gives the same prediction error bars as the exact GP.

3.1.2 Step Function

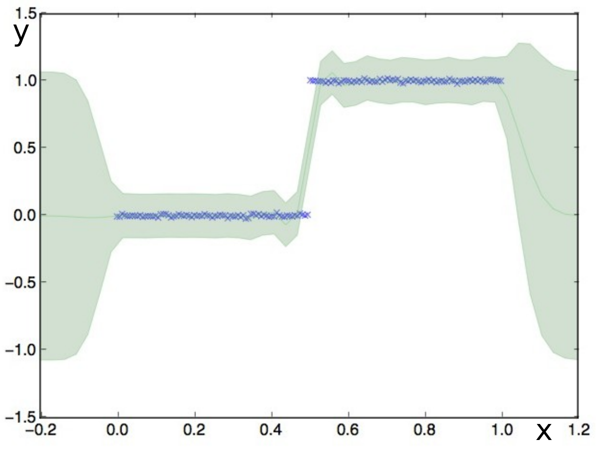


Figure 3.2: Sparse GP Modelling the Step Function

The sparse GP models the mean of most parts of the step function fairly accurately, but is much less accurate for points near to the step-change (completely missing many of the points near the jump). It compensates for its inability to model the discontinuity by putting a relative large variance over all points (so that the penalty over points it misses at the step-change is reduced). Accordingly, the model attains relatively low likelihoods over the test-data (227) 3.1 when compared with the DGP (808).

3.1.3 Optimisation

In the figure below, we display the variational log-likelihood over the test-data during training. We also plot the gradient of the bound, which help us to independently ascertain whether the algorithm converges to a local optimum. The optimisation method uses conjugate gradients, and as can be seen the negative-log-likelihood (from the variational approximation) bottoms out after only 20 or so iterations when training on the sinusoidal data. The model is slower but still fast when training on the step-data, with the variational bound having converged after around 180 iterations.

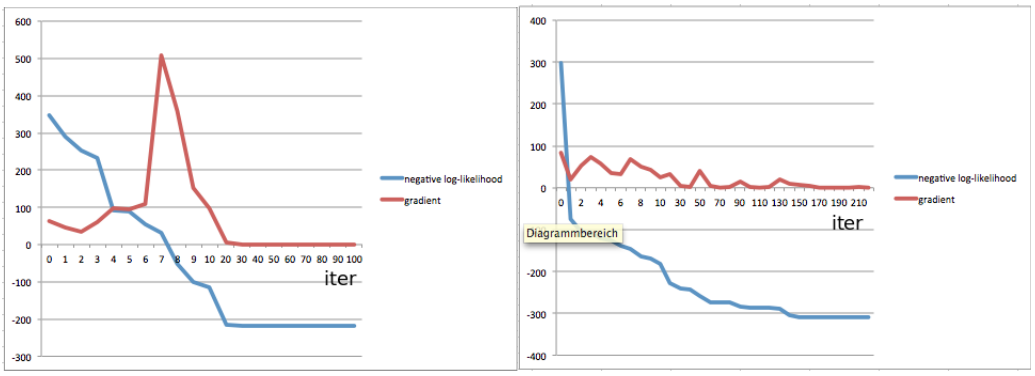


Figure 3.3: GP Training: Left- over Sinusoids; Right- over Step Function

As an evaluation metric, we present the likelihoods over the test and data mean-squared errors for both the step function and the sinusoid.

(Test 321 data-Points)	Step	Sinusoid
likelihood Test	227.17	263.64
Squared-Error	30.11	34.76

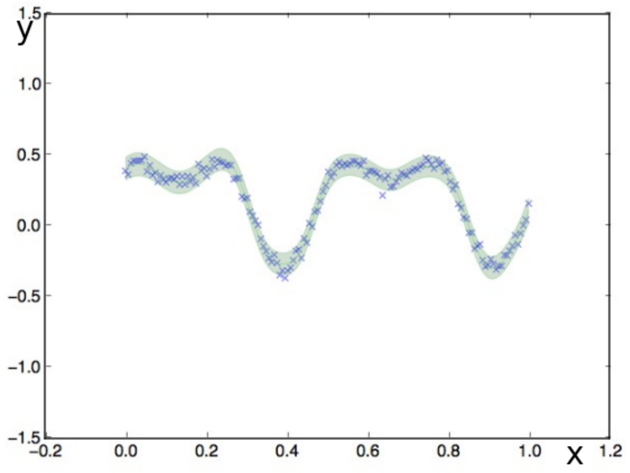
Table 3.1: Sparse GP Regression Results

3.2 Deep GP Results

We next investigate the performance of my Tensorflow DGP implementation, which also uses the optimal $q(u)$ distribution when determining the variational bound on the log likelihood. We focus on a simple 2-layer example, although we note that the implementation may easily be extended to model more layers, since a separate function call is used to compute the part of the bound corresponding to a layer.

3.2.1 Sinusoidal Function

The figures below display the DGP predictions over the test-set as well as samples taken from the DGP.



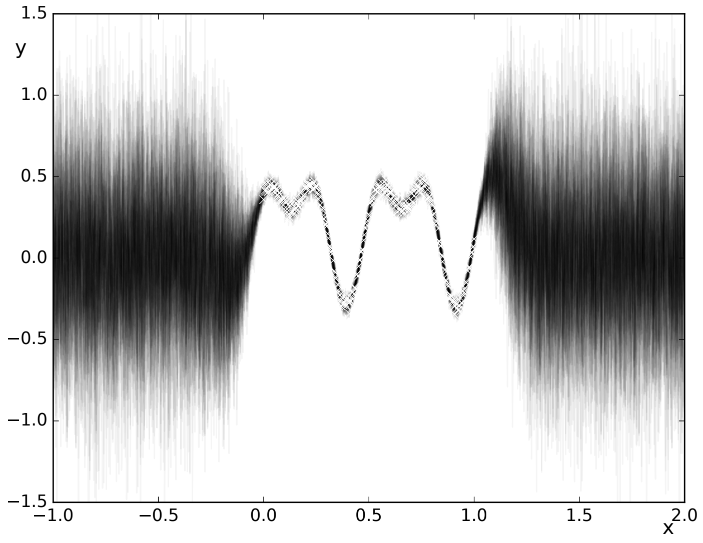
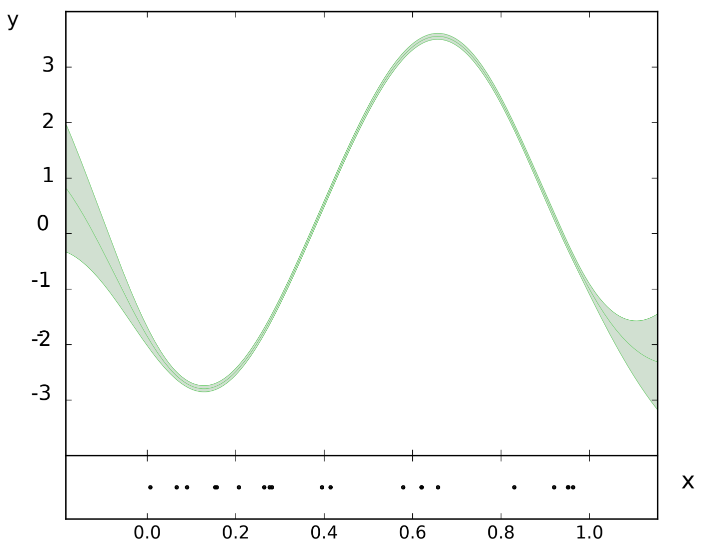


Figure 3.4: Predictive Mean and Error bars (Top) and Samples Drawn from our DGP; The White or Blue Crosses Form the Training Data.

Much like a regular GP, the DGP can model the sinusoidal curve well near the training-data. However the DGP is less confident when it comes to making predictions on data outside the training range. As should be noted, GP's are local and tend to put high uncertainty at places where there is little to no data. With a DGP, the high variance outside the training range comes as a result of uncertainties adding up through the layers.



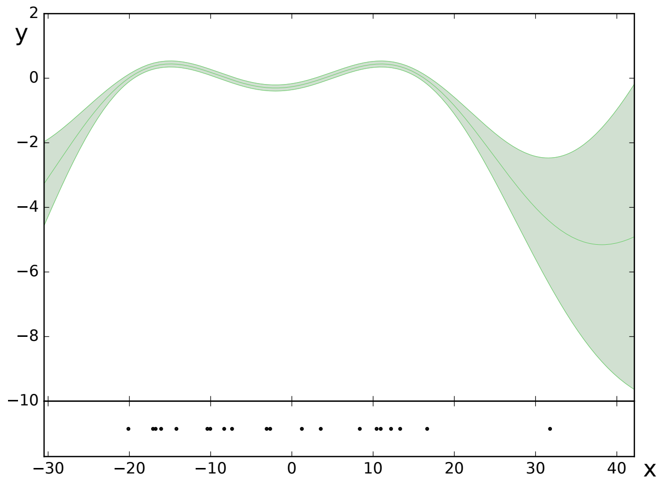


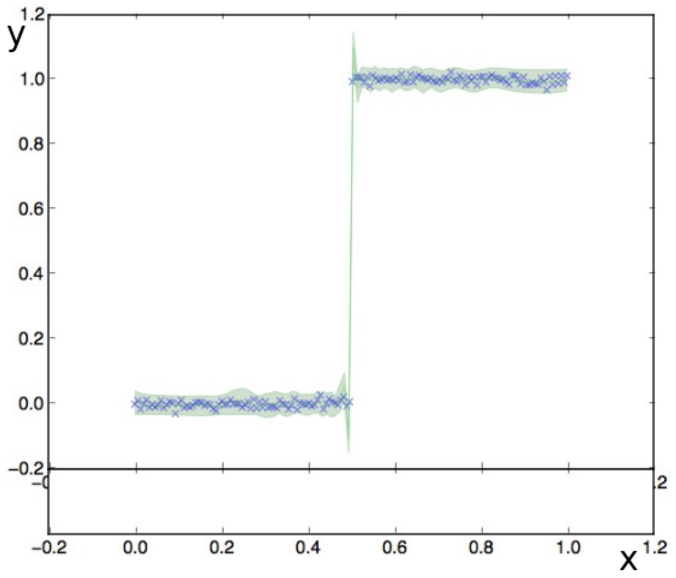
Figure 3.5: Top: Input Layer; Bottom: Output Layer; Pseudo-Points Boxed.

Above, we display the mean prediction and error bars at each hidden layer within the DGP. Pseudo-point inputs are marked with a dot. Interestingly, the output layer of the DGP seems to model one period of the sinusoidal data, whereas the the input layer seems to capture the rest of the periodicity. We discussed, in the introduction, the ability for deep models to use features at different layers through the architecture: here we have a clear example of this in our DGP model.

During my investigations, I produced other plots of the hidden layers in the DGP. For the sinusoid, occasionally the DGP would model a linear function at one of its layers, and effectively model all of the curvature with a single GP (at the other layer). This may well be because the function is well modeled by a single GP (as prior experimentation has shown), and so in this case the extra layer is redundant.

3.2.2 Step Function

Below we display the mean and error bars from the DGPs predictions as well as samples drawn from the DGP, when trained on the step-function data.



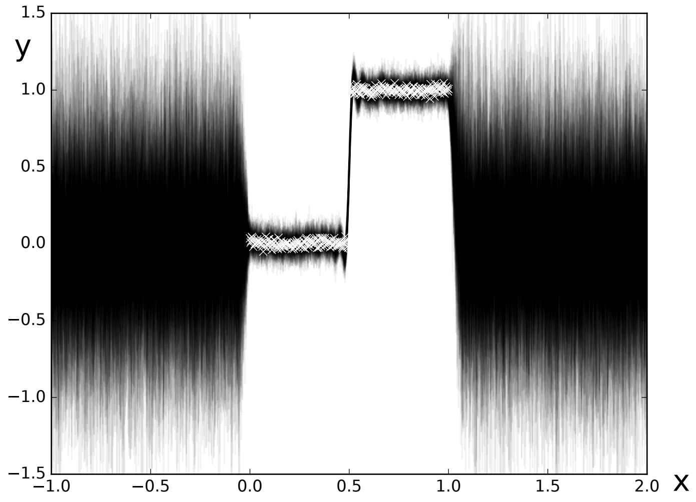


Figure 3.6: Top: Mean Prediction and Error Bars; Bottom Samples Drawn from the Trained Model

Unlike a standard GP, the DGP has the flexibility to model the function more tightly and produces a much steeper vertical rise at the step-change than the ordinary GP. This is due to the fact that the joint normal distribution over derivatives no longer applies to the DGP, enabling it to model functions with steeper and more abrupt jumps.

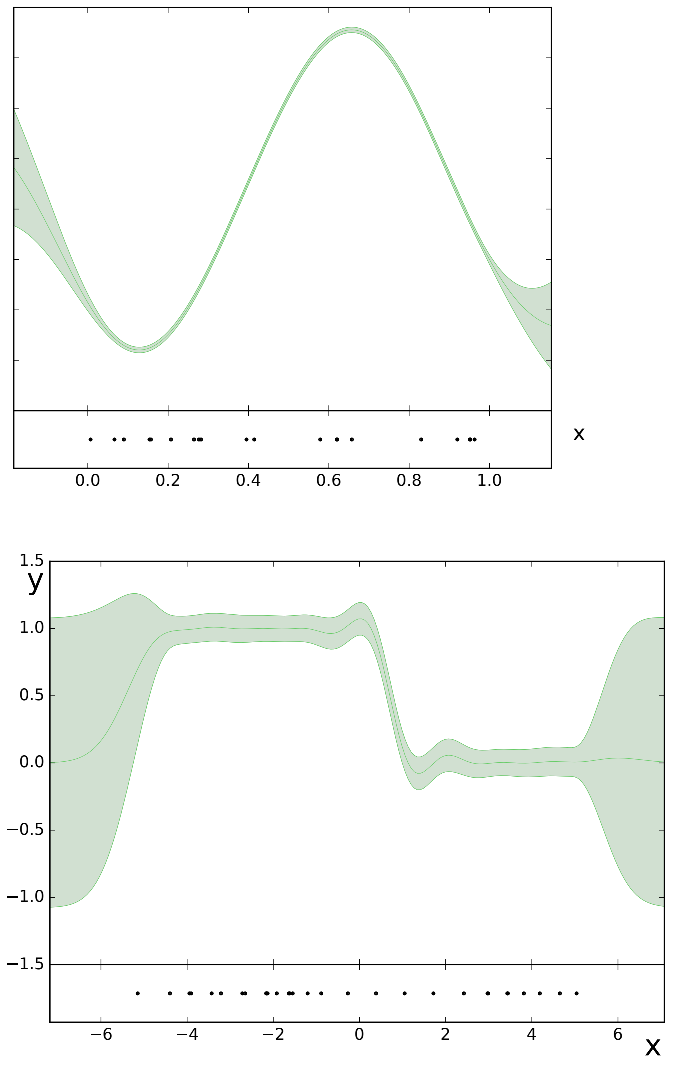


Figure 3.7: Predictions from First (Top) and Second (Bottom) Layers of the DGP. Please note that, although not displayed, the output range for the first layer is coterminous with the input range for the second.

Inspecting the hidden layers, we may see that: at the first layer, the DGP is expansionist and maps the data to a much wider range than that of the inputs. At the second layer, the DGP then compresses the inputs to a much smaller range. This expansion and subsequent compression enables the DGP to model the steep incline of the step-function, using relatively smooth functions at each layer.

3.2.3 Comparisons

The optimisation for the DGP was fast, with a much smoother convergence to the minimum than was managed by the GP (for the step-function data the DGP converges to the optimum 90 iterations quicker).

	Step	Curve
likelihood Test	435.15	286.02
MS -Error	30.16	35.03

Table 3.2: DGP Final Results

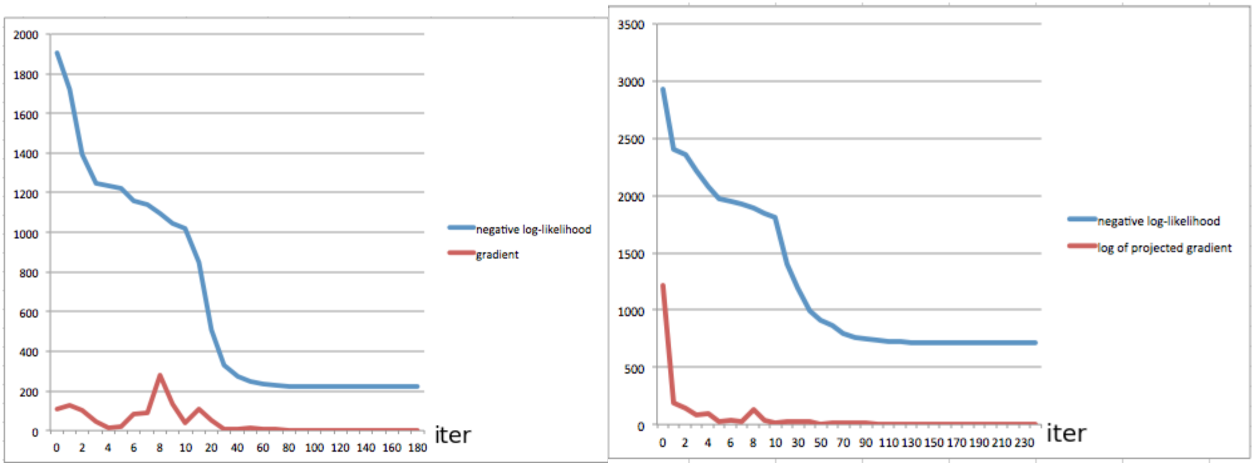


Figure 3.8: Optimisation for the DGP: Let Sinusoids; Right Step Function

When modelling the sinusoidal data, the DGP attained similar mean-squared errors and a slightly increased likelihood over the test-data. These relatively small difference may be put down to the fact that the DGP produced a model that was not too dissimilar from that of an ordinary GP.

With the step-function data, The DGP saw improvements in both the MS error and the likelihoods when modelling the test-data. In fact, there was quite a dramatic improvement in the test-likelihoods (almost twice as high at 405), most likely because the DGP is able to fit error-bars to the training-data much more tightly, whilst 'missing' far fewer points at the step-change. In contrast, the GP tends to inaccurately model quite a few points at the step-change and overcompensates for this by increasing the variance.

3.3 Implementation and Further Extensions

3.3.1 Description of Implementation

The sparse GP implementation was built from scratch and was based upon the Titsias 09 paper and was programmed in Tensorflow.

While programming these models, I produced two separate implementations of a DGP. The first implementation used two separate function calls to compute the variational lower bound at the first and second layers, as well as methods for sampling and predictions. This involved training the model with the test inputs so that it could maintain a distribution over hidden variables $q(h^{L-1})$ for these test inputs. Ultimately output predictions could be made using these variables, according to page 8 of Bui, Li et al [2016]. The DGP implementation seemed to produce somewhat reasonable samples and predictions, but the first-order gradient based optimisation procedure seemed somewhat unstable.

Much later, I discovered a programming package for GPs in Tensorflow, called GPflow. My second implementation borrowed more heavily from GPflow, reusing code from their various GP implementations. A great deal of time was spent getting their GP package to interface with my code. For instance, GP-flow has a special parameter object, and when returning values from functions or initialising data, care must be taken to ensure the right types are returned. In addition to working on the interface, I added functionality to produce the mean and error bars for a DGP, following Bui [2016] as mentioned above. Having done this, I could then use GPflow's more sophisticated second order methods to optimise the DGPs parameters, like conjugate gradients.

All the code written is listed in the appendix.

3.3.2 Further Extensions

It would be relatively straight-forward to extend my my 2-layer DGP implementation in Tensorflow so that it had more layers. This is because the variational lower bound factorises per layer, and so the same function that is used to compute the bound at the second layer may be used to compute the bound at subsequent layers: all one needs do is initialise the relevant variables.

3.4 Challenges Overcome

On our way to building the final DGP implementation, a number of other GP and DGP variations were developed. In particular, we will discuss the results from the natural based optimisation schemes mentioned in the previous chapter. But we do so only while describing the model's progression, the various implementations tried, the challenges faced along the way, and the lessons learnt while striving towards a fully functional DGP model.

3.4.1 First DGP attempt

Our first implementation was based on Damanou's thesis [2015], whose work we greatly appreciate for the clarity of its descriptions. The basic model described was that of a DGP which maintained an explicit distribution over $q(u)$, whose mean and covariance were optimised as variational parameters. Having prorgrammed the model as described in the appendix of Damianou [2015], I discovered a couple of errors (typographical) in the variational bound which led to poor optimisation. Once these had been found and corrected improvements were found. However, for the following reasons we found optimisation difficult:

1. The model maintained a distribution over the inducing points $q(u)$, rather than analytically solving for the optimal distribution.
2. The model did not use second order optimisers like conjugate gradients, and instead relied upon Tensorflow's in-house gradient-based optimisers.

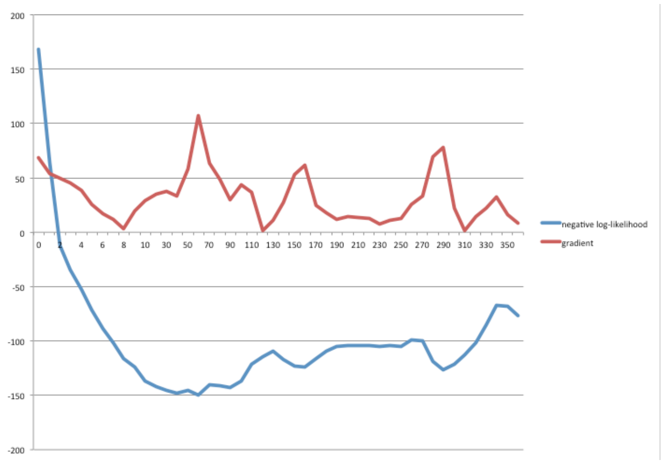


Figure 3.9: Tensorflow Optimisation with Natural Gradients

Below we produce a plot showing the log-likelihood and gradients when optimising under this scheme. As one can see from the graphs, the optimisation procedure was much more erratic, displaying no signs of smooth convergence.

The two things that we could do to remedy the issue were to: (1) improve the optimisation for the $q(u)$ by using natural gradients. From my experimentation, this certainly helped to optimise the mean parameters for the pseudo-points, however it still remained difficult to jointly optimise the other parameters. As an example, take the figure below: the predicted mean given by the model seems to be correct (because the pseudo-points have the correct mean). However the variances and noise terms are not.

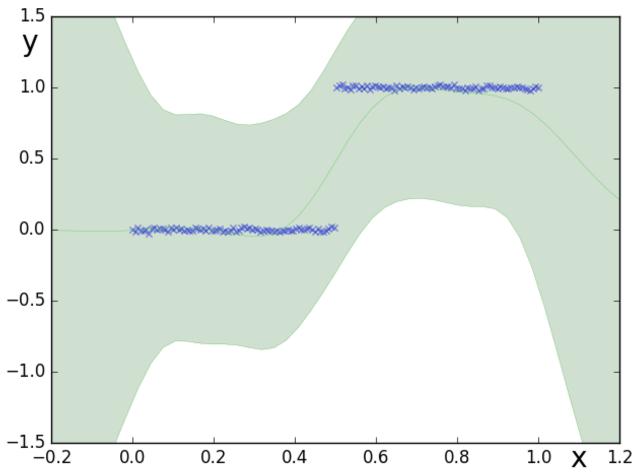


Figure 3.10: Natural Gradient Optimisation

By visualising the pseudo-point means, as above, I was were fairly confident that the natural gradient updates for the means and covariances of $q(u)$ were working as they ought to. Indeed, when the other parameters were all fixed, the natural gradients method would perform well, as an example:

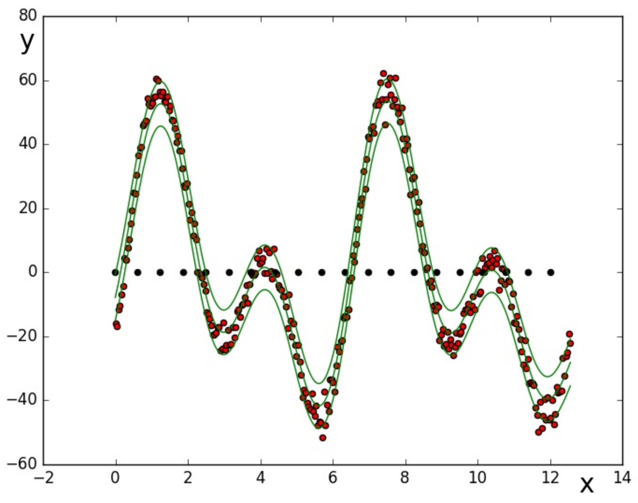


Figure 3.11: Natural Gradient Optimisation, Holding all other Parameters Fixed

However, the trouble was with jointly optimising the other model parameters, for instance length-scales of the Kernel function. Because the natural means were being optimised independently of the pseudo point locations, we could only use a very small step-size. Otherwise, the natural mean updates wold no longer apply to the new pseudo-point locations (as a functions output is highly dependent on the input).

However, with such a small step-size, the other parameters (kernel length-scales etc, function noise etc.) would not be optimised well during training, unless a great many updates were used.

Because of this, I decided to abandon the natural gradient based approach discussed in the preceding section in favour of the Titsias approach which analytically optimises the bound at the optimal variational distribution $q(u)$, completely eliminating the need to perform a challenging joint optimisation problem. This resulted in much improved modelling, as can be seen from the samples taken below from the DGP.

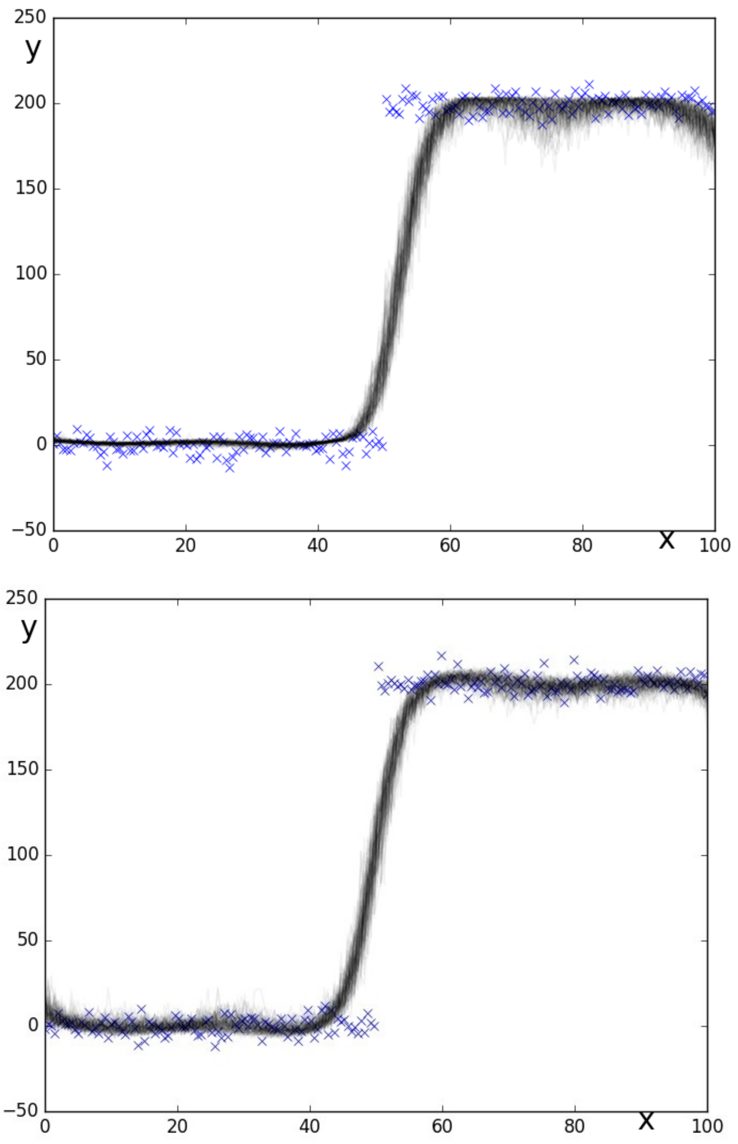


Figure 3.12: 1st DGP Implementation Samples

However, the optimisation was still not as smooth as was hoped and occasionally the variances of the samples would balloon upwards or downwards, as can be seen from the samples above. My hypothesis was that the in-house optimisation method used by Tensorflow was not quite smooth enough. This may be seen from the plots displaying the movement of the gradients below, which appeared to be somewhat erratic, although much reasonable than they were previously, converging in about 200 iterations- about a factor of 3 worse than the conjugate gradients implementation.

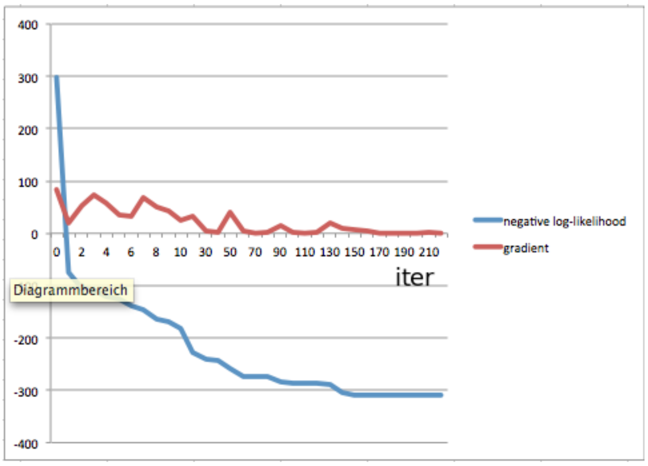


Figure 3.13: 1st DGP Implementation: Optimisation and Training

In my fnal implementation I relied on a powerful second-order method (conjugate gradients) to ensure a fast and smooth optimisation of the model parameters, which converge to an optima in under 60 iterations.

3.5 Conclusions

By implementing these methods, we discovered that: DGP implementations benefit from using a bound evaluated at the optimal $q(u)$, as specified by Titsias [2009]. Not only does this reduce the number of variational parameters, but it also tightens the variational approximation to the log-likelihood. The use of conjugate gradients certainly helps to hasten and smoothen optimisation when compared with Tensorflow's in-house (gradient-descent) optimisers. I also learned separately, that ensuring the matrix algebra is performed in a numerically stable manner is a must when working with DGPs: for instance noise must be added to Cholesky decompositions and matrix inversions must be handled very carefully.

Our contributions this chapter have produced a working Tensorflow implementation of a DGP that accurately models the step function, and can be extended to higher dimensions and more layers with relative ease. Although the optimisation is done in numpy using CG, as the Tensorflow library matures, and more sophisticated optimisations schemes become available, it should be possible to run the model in its entirety in Tensorflow. Ultimately, we hope model will be of some benefit to practitioners who hope to speed-up their experimentation using GPUs.

Chapter 4

Deep GP Variational Inference Extensions

4.1 Executive Summary

- Variational inference in a DGP uses a variational distribution $q(u, h)$ to model the the hidden variables u, h (for the inducing points and hidden layers) within the model to form a lower bound of the log-likelihood.

- In a DGP, the variational distribution $q(u, h)$ is taken to be mean-field over many of the variables of interest: $q(u, h) = \prod_{ld} q(u^{ld}) \left[\prod_{n} q(h_n^{ld})\right]$. This means that the variational distribution over inducing outputs u is factored by layer and dimension; that the distribution over h variables is factored by layer, point index and dimension (i.e completely mean-field); and also that the inducing outputs u can be modeled separately from h. These are all modelling assumptions that may be challenged, leading to richer variational approximations and consequently tighter bounds on the log-likelihood.

- As depicted below, variational inference may be interpreted as proposing a family of distributions Q that are used to approximate the underlying probability distribution $p \in P$. We obtain a better approximation to p by minimising the KL-divergence, $\mathrm{KL}(q|p)$, represented by the arrows in the diagram. Ultimately, by modelling more of the dependencies between variables in a DGP, we obtain a richer family of distributions Q, which means that it is possible to find a single $q \in Q$ that better approximates the log likelihood as specified by our model p. For further reference see [Wainright and Jordan 2009] .

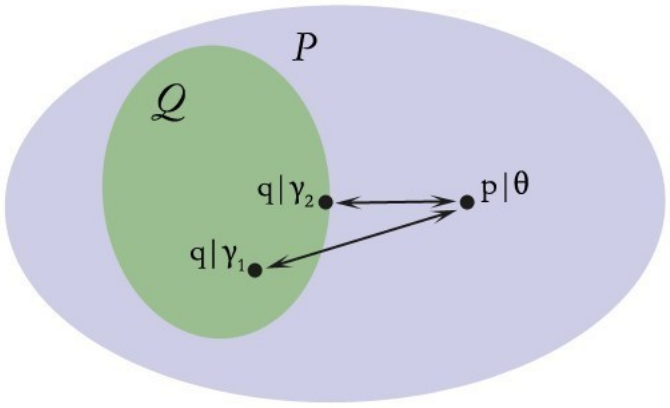


Figure 4.1: Variational Inference between Probability Families: Arrows Denote KL Divergence.

- In this chapter, we will look at ways to model an increased number of dependencies in DGPs. We will first consider (1) modelling the dependencies of the h variables on pseudo inputs u at a given layer, (as a shorthand) $q(h_n^{ld}|u^{ld})$. (2) After this, we extend the method to model dependencies between dimensions $q(h_n^l|u^l)$. We will then discuss how to model dependencies between layers. In all cases we model the dependencies between variables as first-order Markov Gaussian.

- We will aim for a variational approximation that is tractable, which we define as: computable in $O(nm^2d^2l)$, [i.e. the same compute time for a fully connected RBM, pictured below]

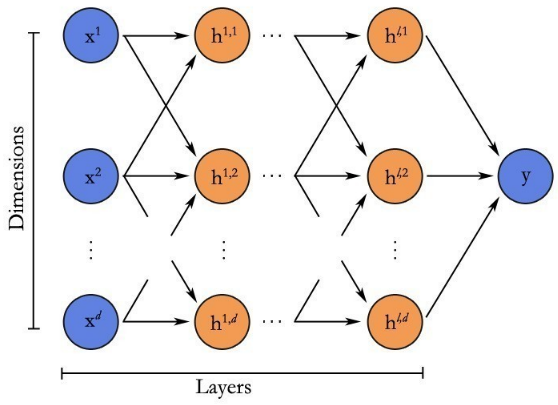


Figure 4.2: Fully Connected Restricted Boltzman Machine. As can be seen from the diagram, there are $O(d^2l)$ connections, which means at each iteration a gradient based optimisation scheme over n data-points will cost $O(nd^2l)$.

4.2 Framework:

In the next chapter, we will propose a framework for our investigations that follows three simple steps.

1. Write down a first form for the variational lower bound, which has yet to be refined.
2. Find the distribution for the optimal $q(u)$ based on the form of the bound in (1).
3. Evaluate the variational lower bound at the optimal $q(u)$.

Once we have done this, using the distribution $q(u)$ we can take samples from our model and make predictions at each hidden layer, much in the same as the procedure for ordinary DGPs.

We present graphical models for both the DGP and the variational approximation below, as a succinct reminder of the modelling assumptions. Note that for the variational distribution's graphic, the red boxes do not represent factored modelling assumptions as they do for ordinary graphical models; rather they express the opposite- that we are modelling a set of variables jointly.

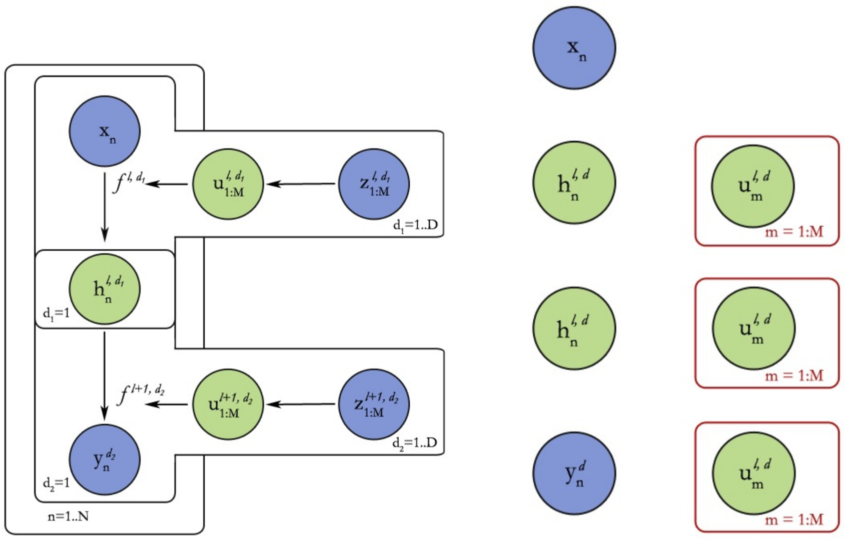


Figure 4.3: Left: DGP Model; Right: Basic Variational Modelling Assumptions

---

4.3 Dependencies on Inducing Points $q(h_n^{ld}|u^{ld})$:

Explicitly considering the distribution over f^{ld}, our variational approximation will take the form:

$$q(h,f) = \prod_{l,d} \left[p(f^{ld} \neq u^{ld}|u^{ld}) q(u^{ld}) \prod_{n} q(h_n^{ld}|u^{ld}) \right]$$

The only change that we make to the variational distribution is to model $q(h_n^{ld})$ as $q(h_n^{ld}|u^{ld})$, which means that for a given dimension and layer, we model dependencies between h_n^{ld} and its inducing points. Just as in the ordinary DGP variational distribution, we model the m points u^{ld} jointly for a given layer and dimension. These new modelling assumptions are pictured in the diagram below:

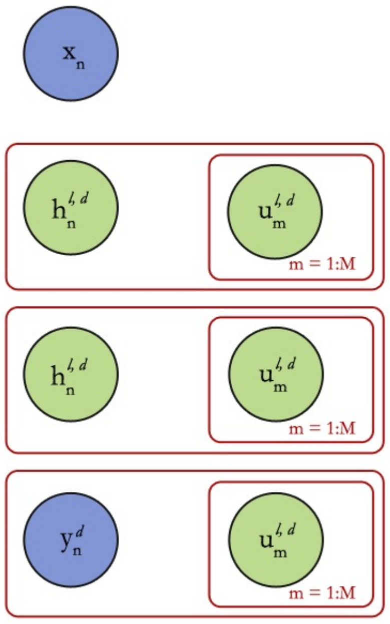


Figure 4.4: Variational Approximation with Dependencies on Inducing Points

The distribution for $q(h_n^{ld}|u^{ld})$ is modelled as:

$$q(h_n^{ld}|u^{ld}) = N(h_n^{ld}|\mathcal{T}_n^{ld}u^{ld} + \mathcal{M}_n^{ld}, \mathcal{S}_n^{ld})$$

This corresponds to a Markov model, with dynamics:

$$h_n^{ld} = \mathcal{T}_n^{l}u^{ld} + \mathcal{M}_n^{ld} + \epsilon_n^{ld}\ , \epsilon_n^{ld} \sim N(0, \mathcal{S}_n^{ld})$$

Here ϵ_n^{ld} is modelled by a univariate Gaussian, $\mathcal{M}^{ld}$ is univariate and $\mathcal{T}^{ld}$ is a $1 \times m$ transition matrix. To improve our notation, we may re-write these equations as multivariate Gaussians by grouping the variables h according to their index n. Then:

$$h^{ld} = \mathcal{T}^{ld}u^{ld} + \mathcal{M}^{ld} + \mathcal{S}^{ld}\ , \text{ with } \mathcal{S}^{ld} \text{ diagonal.}$$

Where $\mathcal{M}^{ld}$ and $\mathcal{T}^{ld}$, and $\mathcal{S}^{ld}$ are $n \times 1$, $n \times m$ and $n \times n$ diagonal respectively.

4.3.1 Merits and Disadvantages of the new Distribution

The number of variational parameters has risen from $O(n)$ to $O(nm)$, which may make optimisation more difficult. However, since m should be small relative to n, we hope that this fact will not prove too troubling in practice. I believe this modelling setup has the potential to greatly improve upon the standard mean-field approach. In Titsias' original paper, u^{ld} was described a sufficient statistic for$f^{ld} \neq u^{ld}$, which is just a noise-free h_n^{ld}, so we would expect there to be strong dependencies between the two that could be captured by moving from mean-field modelling. What's more, given such a model, it is still possible to apply Titsias' method to find the optimal distribution $q(u^{ld})$.

4.3.2 Writing the variational Lower bound

Following similar procedures to those described in chapter 2, the lower bound may be written as:

$$F(q) = \sum_{l,d} \langle \mathcal{E}^{ld} \rangle_{q(h,u)} - \sum_{l,d} \mathrm{KL}(q(u^{ld}) | p(u^{ld})) + \sum_{l,d} \int q(u^{ld}) \mathrm{H}(q(h^{ld}|u^{ld}))$$

Here we may think of $Y = h^{Ld}$ and $X = h^{0d}$ as variables with zero variance We further note that the indices l on the sums are taken over $1 : L-1$ and that $\mathcal{E}^{ld}$ is given by[1]:

$$\mathcal{E}^{ld} = \langle \log[p(h^{ld}|f^{ld} \neq u^{ld}, h^{l-1d})] \rangle_{q(f^{ld}|u^{ld})}$$

We may factorise the terms that depend on u in this bound with respect to ld. We do so, dropping all indices on l, d for ease of notation.

$$F(q) = \langle \mathcal{E} \rangle_{q(h,u)} - \int q(u) \log \frac{q(u)}{p(u)} + \int q(u) \mathrm{H}\left[q(h|u)\right] \tag{4.1}$$

4.3.3 Determining the Optimal distribution $q(u)$

To obtain the optimal $q(u)$, we must differentiate $F(q)$ but with the added Lagrangian constraint, to ensure $\int q(u) = 1$. We do this to obtain:

$$\frac{\delta}{\delta q(u)} \left[F(q) - \lambda\left(\int q(u) - 1\right) \right] =$$

$$= \langle \mathcal{E} \rangle_{q(h|u)} - \log q(u) + \log p(u) + \mathrm{H}\left[q(h|u)\right] - \lambda - 1 \tag{4.2}$$

Note that the entropy $\mathrm{H}\left[q(h|u)\right]$ of a distribution may be written as follows:

$$\mathrm{H}\left[q(h|u)\right] = \frac{n}{2}(1 + ln(2\pi)) + \frac{1}{2} ln|\mathcal{S}|$$

As such, the entropy $\mathrm{H}\left[q(h|u)\right]$ is constant with respect to the variable u , which means that the optimal distribution $q(u)$ takes the form:

$$= \langle \mathcal{E} \rangle_{q(h|u)} - \log q(u) + \log p(u) + constants \tag{4.3}$$

This is precisely the same form as in chapter 2:

[1] $\mathcal{E}^{ld}$ log probability of a conditional Gaussian with noise

$$q(u) \propto e^{\langle \mathcal{E} \rangle_{q(h|u)}} p(u)$$

4.3.3.1 Finding Parameters for $q(u)$ by Unpacking $\langle \mathcal{E} \rangle_{q(h|u)}$

By inspection, we may see that $q(u)$ is Gaussian. So it remains to find the mean and covariance of the distribution (the sufficient statistics). We will do this by unpacking the $\langle \mathcal{E} \rangle_{q(h|u)}$ term.

As has been demonstrated in chapter (2), considering the terms that depend on u, we may write:

$$\langle \mathcal{E} \rangle_{q(h^-|u)} = -\frac{1}{2\sigma^2}\left[h^T h - 2(h^T \langle K_{nm} \rangle u) + (u^T K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle K_{mm}^{-1} u)\right] + constants$$

Here we have used the following short-hands: $h^- = h^{l-1d}$ and $\langle K_{mn} K_{nm} \rangle = \langle K_{mn} K_{nm} \rangle_{q(h^-)}$.

We further evaluate $\langle \mathcal{E} \rangle_{q(h^-|u)}$ by applying the Markov transition dynamics, i.e. by writing:

$$h = \mathcal{T} u + \mathcal{M} + \epsilon, \text{ where } \epsilon \sim N(0, \mathcal{S})$$

We also make use of the (Markov Modelling 1) result contained in the appendix, our expression evaluates to:

$$\langle \mathcal{E} \rangle_{q(h|u)} = -\frac{1}{2\sigma^2}\left[u^T \mathcal{T}^T \mathcal{T} u + 2u^T \mathcal{T}^T \mathcal{M} + Tr[\mathcal{S}] - 2(\mathcal{T} u + \mathcal{M})^T \langle K_{nm} \rangle K_{mm}^{-1} u)\right]$$
$$-\frac{1}{2\sigma^2}(u^T K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle K_{mm}^{-1} u)$$

Given this, to find the optimal distribution $q(u)$, which we know is Gaussian, we need only expand the quadratic in u, given by: $\langle \mathcal{E} \rangle_{q(h,h^-|u)} + \log p(u)$. Having done so, and grouped terms together, we discover:

$$\Sigma^{-1} = \frac{1}{\sigma^2}\left[\mathcal{T}^T \mathcal{T} - 2\mathcal{T}^T \langle K_{nm} \rangle K_{mm}^{-1} + K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle K_{mm}^{-1}\right] + K_{mm}^{-1}$$

$$\mu = \Sigma \frac{1}{\sigma^2}\left[K_{mm}^{-1} \langle K_{mn} \rangle \mathcal{M} - \mathcal{T}^T \mathcal{M}\right]$$

This determines the distribution over $q(u)$. For completeness, note that a repeated application of the matrix inversion lemma will show that Σ^{-1} is invertible.

4.3.4 Determining $\mathcal{E}_{q(h,h^-|u)}$ and $\langle \mathcal{E}_{q(h,h^-|u)} \rangle_{q(u)}$

To evaluate the bound, it will be necessary to compute the value $\langle \mathcal{E}_{q(h,h^-|u)} \rangle_{q(u)}$. We do this by following a similar procedure to that in the DGP section of chapter 2: we first write the expression $\mathcal{E}_{q(h,h^-|u)}$, taking $Q_{mm} = K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle$:

$$\mathcal{E}_{q(h,h^-|u)} =$$

$$logN(0;0,\sigma^2 I_n) - \frac{1}{2\sigma^2}\left[Tr(\langle K_{nn}\rangle) - Tr(Q_{mm})\right]$$

$$-\frac{1}{2\sigma^2}\langle u^T(\Sigma^{-1} - K_{mm}^{-1})u - 2u^T\Sigma^{-1}\mu + Tr[S]\rangle$$

Taking the expectation of the above expression over $q(u)$, we determine that [2]:

$$\langle\mathcal{E}_{q(h,h^-|u)}\rangle_{q(u)=}$$

$$logN(0;0,\sigma^2 I_n) - \frac{1}{2\sigma^2}\left[Tr(K_{nn}) - Tr(Q_{mm})\right]$$

$$-\frac{1}{2\sigma^2}\langle Tr[(\Sigma^{-1} - K_{mm}^{-1})(\Sigma + \mu^T\mu) - 2\mu^T\Sigma^{-1}\mu + Tr[S]\rangle$$

4.3.5 Variational lower bound:

Recall from equation (1) that the part of the bound relating to dimensions l and d may be written as:

$$F(q) = \int_u q(u)\langle\mathcal{E}\rangle_{q(h,h^-|u)} - \mathrm{KL}\left[q(u)|p(u)\right] + \int q(u)\mathrm{H}\left[q(h|u)\right]$$

We note that:

- $q(u)\sim N(\mu, \Sigma)$
- $q(h|u) = N(\mathcal{T}u + \mathcal{M}, \mathcal{S})$, with $\mathcal{S}$ diagonal.
- $p(u) = N(0, K_{mm})$

In working towards this section, we have already evaluated all of these expressions, except the KL which may be evaluated using standard formulas for MVNs. To write the variational bound in its entirety, we have:

$$F(q) =$$

$$logN(0;0,\sigma^2 I) - \frac{1}{2\sigma^2}\left[Tr(K_{nn}) - Tr(Q_{mm})\right]$$

[2]To take the expectation, we apply the trace-trick $x^TAx = Tr[Axx^T]$

$$-\frac{1}{2\sigma^2}\langle Tr[(\Sigma^{-1} - K_{mm}^{-1})(\Sigma + \mu^T\mu) - 2\mu^T\Sigma^{-1}\mu + Tr[S])$$

$$-\mathrm{KL}\left[N(\mu, \Sigma)|N(0, K_{mm})\right] + \mathrm{H}\left[N(0, \mathcal{S})\right]$$

As a final remark, we note that as the bound factors in n, l, d , with the most costly operation at each layer and dimension being the $O(nm)$ matrix multiplications. Thus, the compute time and number of variational parameters may be given as:

Compute Time	Variational parameters
O(ldm^2n)	O$(lmnd)$

4.4 Dependencies between dimensions: $q(h^l|u^{l-1})$:

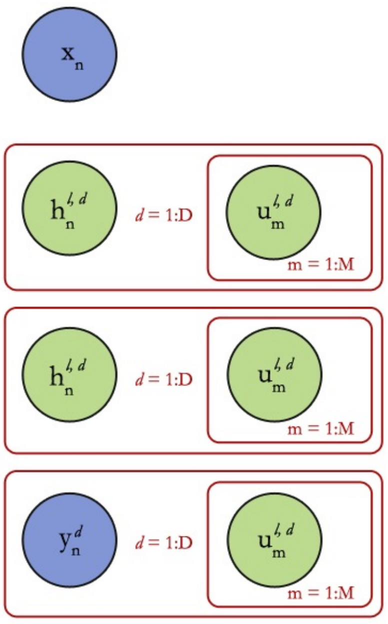


Figure 4.5: Variational Graphical Model With Dimensions Dependencies between Dimension and between h and u

The previous method induces dependencies between h and u, however the bound remains factored by dimension. We will now look at modelling the dimensions of h and u together. The only change we make to the variational model is that it is no longer factored by dimension, and so it now takes the form:

$$q(h, f) = \prod_l p(f^l \neq u^l|u^l)q(u^l)\prod_n q(h_n^l|u^l)$$

Here the Markov modelling assumption becomes:

$$h_n^l = \mathcal{T}_n^l u^l + \mathcal{M}_n^l + \epsilon_n^l \text{where } \epsilon_n^l \sim N(0, \mathcal{S})$$

Here $\mathcal{S}_n^l$, $\mathcal{T}_n^l$, $\mathcal{M}_n^l$ are $d \times d$, $d \times md$ and $d \times 1$ matrices respectively .

The steps we must take to determine the variational lower bound are as before:

1. Write the preliminary variational lower bound
2. Find the distribution for the optimal $q(u^l)$
3. Evaluate the bound at the optimal $q(u^l)$

4.4.1 Writing the preliminary variational Lower bound

Following similar procedures to those in the previous section, the bound may be written as:

$$F(q) = \sum_{l,d,n} \langle \mathcal{E}_n^{ld} \rangle_{q(h,u)} - \sum_l \mathrm{KL}(q(u^l)|p(u^l)) + \sum_{l,n} q(u)\mathrm{H}\left[q(h_n|u)\right] \tag{4.4}$$

Here we take $p(u^l) = \prod_d p(u^{ld})$ since p factors by dimension d.

4.4.2 Determining the Optimal distribution $q(u^l)$

For ease of notation, we drop subscripts l, since the bound factors in l. By differentiate expression (4) with respect to $q(u)$, and equating to 0, we obtain:

$$q(u) \propto e^{\sum_{d,n} \langle \mathcal{E}_n^d \rangle_{q(h|u)}} p(u)$$

We now turn to evaluating the mean and covariance of $q(u)$, by unpacking $e^{\langle \mathcal{E} \rangle_{q(h|u)}}$ much like in the preceding section. Here we take K_{nm} to be a $1 \times m$ matrix, for the n'th data-point, and similarly for K_{nn}. We now have:

$$\sum_{d,n} \langle \mathcal{E}^{d,n} \rangle_{q(h|u)} = \sum_{d,n} -\frac{1}{2\sigma^2}\left[u^T \mathcal{T}_n^T \mathcal{T}_n u + 2u^T \mathcal{T}_n^T \mathcal{M}_n + Tr[\mathcal{S}_n] - 2(\mathcal{T}_n u + \mathcal{M}_n)^T \langle K_{nm}^d \rangle (K_{mm}^d)^{-1} u)\right]$$

$$\sum_{d,n} -\frac{1}{2\sigma^2}(u^T (K_{mm}^d)^{-1} \langle K_{mn}^d K_{nm}^d \rangle (K_{mm}^d)^{-1} u)$$

We can group all of the terms in the equation above to be of the form: $-\frac{1}{2} u^T \gamma_n^d u - u^T \beta_n^d$, for expressions γ_n^d, β_n^d.

Given this, to find the optimal distribution $q(u)$, which we know is Gaussian, we need only unpack$\sum_{d,n} \langle \mathcal{E} \rangle_{q(h,h^-|u)} + \log p(u)$.

Let K be the matrix where:

$$\begin{pmatrix} K^1_{mm} & 0 & 0 & 0 \\ 0 & K^2_{mm} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & K^d_{mm} \end{pmatrix}$$

Then grouping all terms in $\sum\limits_{d,n} \langle \mathcal{E} \rangle_{q(h,h^-|u)} + \log p(u)$ together, we obtain:

$$-\frac{1}{2} u^T \left[\sum_{d,n} \gamma_n^d + K^{-1} \right] u - \sum_{d,n} \beta_n^d$$

Given this, we know the parameters for the optimal $q(u)$ distribution will be:

$$\Sigma^{-1} = \sum_{d,n} \gamma_n^d + K^{-1}$$

$$\mu = \Sigma \sum_{d,n} \beta_n^d$$

4.4.3 Variational lower bound:

Recall from equation (1) that the part of the bound pertaining to the index l may be written as:

$$F(q) = \int_u \sum_{d,n} \langle \mathcal{E}_n^d \rangle_{q(h,h^-|u)} - \mathrm{KL}\left[q(u) | p(u) \right] + \sum_n \int q(u^l) \mathrm{H}\left[q(h_n | u^l) \right]$$

We note that:

- $q(u) \sim N(\mu, \Sigma)$
- $q(h_n | u^l) = N(\mathcal{T}_n u + m_n, S_n)$
- $p(u) = N(0, K)$

Given these formulae, the KL-terms may be evaluated according to standard formulas for MVNs. All that remains is to evaluate the $\sum\limits_{d,n} \langle \mathcal{E}_n^d \rangle_{q(h,h^- u)}$ expectation. Much like in the previous section, this term may be evaluated by recalling that :

$$\sum_{d,n} \langle \mathcal{E}_n^d \rangle_{q(h,h^-|u)} = -\frac{1}{2\sigma^2} u^T (\Sigma^{-1} - K) u - \frac{1}{\sigma^2} u^T \Sigma^{-1} \mu + constants$$

Given this, we may write the expectation explicitly including those terms constant in u as:

$$\sum_d log N(0; 0, \sigma^2 I_n) - \sum_d \frac{1}{2\sigma^2} \left[Tr(K^d_{nn}) - Tr(Q^d_{mm}) \right]$$

$$-\frac{1}{2\sigma^2}\left[Tr\left[\left(\Sigma^{-1}-K^{-1}\right)\left(\Sigma+\mu^T\mu\right)\right]-2\mu^T\Sigma^{-1}\mu+\sum_n Tr[S_n]\right]$$

As a final remark, we note the complexity of the method. At each layer, the most costly operation is first computing Σ^{-1} and then inverting it. A careful analysis shows that computing Σ^{-1} will cost $O(nm^2d^3)$, due to the $\mathcal{T}_n^T\mathcal{T}_n$ terms in the sum, which are $d \times md$ matrices. However, evaluating Σ will cost $O(m^3d^3)$ since Σ is $md \times md$. The number of variational parameters is now $O(md^2nl)$:

Compute Time	Variational parameters
$O(lnm^2d^3)$	$O(lnmd^2)$

These increases should be acceptable for low-dimensional data, however for higher dimensional data the $O(d^3)$ compute time would make this method infeasible.

4.5 Dependencies between Layers $q(u^l|u^{l-1})$:

We now turn to the case where the variational bound includes dependencies between all layers u^l but is factored between u and h. This case is depicted in the figure below.

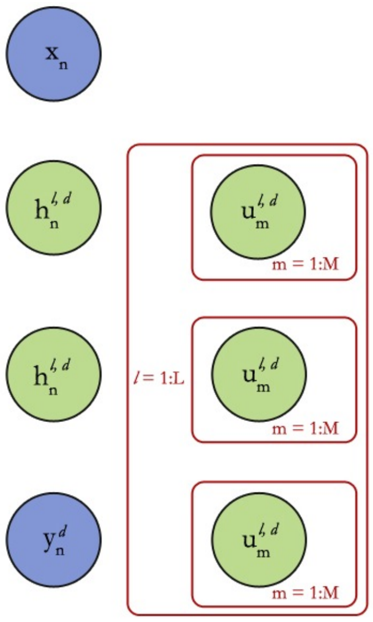


Figure 4.6: Dependencies Across the Layers of u

In this case, the optimal $q(u)$ distribution would be no different from the optimal distribution $q(u)$ in the standard mean-field DGP bound. This means there would be no benefit from this variational modeling assumption. The reason this is so, is that there are no direct interactions between u^l terms in our DGP model; rather the interactions between variables are always intermediated through the hidden variables h^l. And so, the variational distribution q does not have enough 'reach' to capture useful dependencies expressed within the DGP.

To demonstrate that this is the case, consider the bound that would obtain from this variational modelling assumption:

$$F(q) = \sum_{l,d} \langle \mathcal{E}_n^{ld} \rangle_{q(h,u)} - q(u) \log \frac{q(u)}{\prod\limits_{ld} p(u^{ld})} + \sum_{l,d} \mathrm{H}\left[q(h_n)\right]$$

We now take the derivative of the Lagrangian of $F(q)$ and set it to 0:

$$\frac{\delta}{\delta q(u)} \Big[F(q) - \lambda(\int q(u) - 1)\Big]$$

This evaluates to:

$$\sum_{l,d} \langle \mathcal{E}_n^{ld} \rangle_{q(h,u)} - \log q(u) - \sum_{l,d} p(u^{ld}) + constants = 0 \tag{4.5}$$

which means $q(u) \propto \prod\limits_{ld} [e^{\langle \mathcal{E}_n^{ld} \rangle_{q(h,u)}} p(u^{ld})]$. As a result, the distribution is fully factored in u^{ld} terms, and in fact $q(u) = \prod\limits_{ld} q^*(u^{ld})$ where q^* is the optimal mean-field distribution.

4.6 Dependencies between Layers $q(h^{ld}|h^{l-1})$:

In this case, our mean-field variational approximation to the lower bound will take the form:

$$q(h, f) = \prod_{ld} p(f^{ld} \neq u^{ld} | u^{ld}) q(u^{ld}) \prod_{n} q(h_n^{ld} | h_n^{l-1})$$

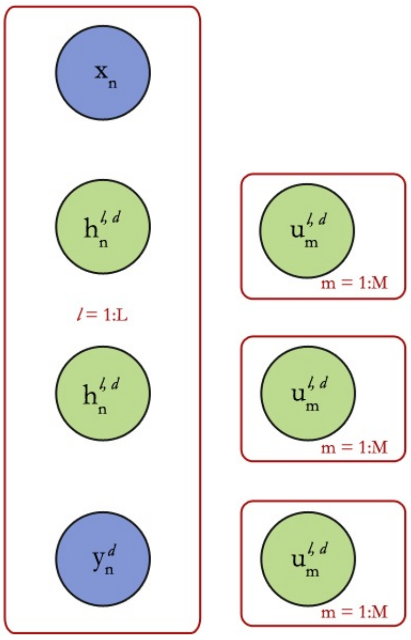


Figure 4.7: Dependencies Modelled Across Layers in h

Thus, the only change we make to the variational model is to model $q(h_n^{ld}|h^{l-1})$ as:

$$h_n^{ld} = \mathcal{T}_n^{ld} h_n^{l-1} + \mathcal{M}_n^{ld} + \epsilon_n^{ld}, \text{ with } \epsilon_n^{ld} \sim N(0, \mathcal{S}_n^{ld})$$

In this setting, $\mathcal{T}_n^l, \mathcal{M}_n^l, \mathcal{S}_n^l$ are respectively $1 \times d$,1×1 and 1×1 matrices.
We now perform the familiar three steps:

1. Write the variational lower bound
2. Find the distribution for the optimal $q(u^{ld})$
3. Evaluate the variational lower bound at $q(u^{ld})$

4.6.1 Writing the variational Lower bound

Following similar procedures to those described in chapter (2), the bound may be written as:

$$F(q) = \sum_{l,d} \langle \mathcal{E}^{ld} \rangle_{q(h)q(u)} - \sum_{l,d} \mathrm{KL}(q(u^{ld})|p(u^{ld})) + \sum_{l,d,n} \int q(h_n^{l-1}) \mathrm{H}(q(h_n^{ld}|h_n^{l-1}))$$

We can factorise the terms in this bound with respect to l, d and drop all indices on l, d for ease of notation. We use the symbol h_n^- to denote h_n^{l-1}. Then:

$$F(q) = \langle \mathcal{E} \rangle_{q(h)q(u)} - \int \mathrm{KL}(q|p) + \sum_n \int_{h_n^-} q(h_n^-) \mathrm{H}(q(h_n|h_n^-)) \tag{4.6}$$

This approximation currently looks very similar to the mean-field bound perhaps because it is. The main difference comes in to play with the expectation $\langle \mathcal{E} \rangle_{q(h)q(u)}$ and the entropy term.

4.6.2 Determining the Optimal distribution $q(u^{ld})$

We can skip a few steps here. We know that the optimal distribution $q(u)$ will take the same form as the mean field bound, due to the similarities in the form of $F(q)$, with the only measurable difference will in the expectation: $\langle \mathcal{E} \rangle_{q(h,h^-)q(u)}$. Thus:

$$q(u) \propto e^{\langle \mathcal{E} \rangle_{q(h|u)}} p(u)$$

Considering only the expressions that depend on u, $\langle \mathcal{E} \rangle_{q(h,h^-)}$ may be written as:

$$\langle \mathcal{E} \rangle_{q(h,h^-)} = -\frac{1}{2\sigma^2} \left[\langle -2(u^T K_{mm}^{-1} \langle K_{mn} h \rangle) + (u^T K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle K_{mm}^{-1} u) \right] - \frac{1}{2} u^T K_{mm}^{-1} u + constnant$$

Given the terms above, we may write the distribution for $q(u)$ as:

$$\Sigma^{-1} = \frac{1}{\sigma^2} \left[K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle K_{mm}^{-1} \right] + K_{mm}^{-1}$$

$$\mu = \Sigma \frac{1}{\sigma^2} \left[K_{mm}^{-1} \langle K_{mn} h \rangle \right]$$

4.6.3 Variational lower bound:

Recall from equation 4.6 that the bound pertaining to dimensions l and d may be written as:

$$F(q) = \langle \mathcal{E} \rangle_{q(h)q(u)} - \mathrm{KL}(q(u)|p(u)) + \sum_n \int_{h_n^-} q(h_n^-) \mathrm{H}(q(h_n|h_n^-))$$

We note that:

- $q(u) \sim N(\mu, \Sigma)$
- $q(h_n) = N(\mathcal{T}_n \mu_{h^-} + \mathcal{M}_n, \mathcal{T}_n \Sigma_{h^-} \mathcal{T}_n^T + S_n)$
- $p(u) = N(0, K_{mm})$
- $p(h_n) = N(0, \underbrace{K_{nn}}_{1\times 1})$

Given this, the KL and Entropy terms are easy to evaluate (Please see section 1.7.4 for an analagous evaluation of the entropy term). All that remains is to evaluate the $\mathcal{E}$ expectation. Let us define:

$$\mathcal{H}_{nm} = \langle h^T K_{nm} \rangle K_{mm}^{-1}$$

$$\alpha_{nm} = \langle K_{nm} \rangle K_{mm}^{-1}$$

$$Q_{mm} = K_{mm}^{-1} \langle K_{mn} K_{nm} \rangle$$

Then, given work from the preceding sections:

$$\int_u \langle \mathcal{E} \rangle_{q(h,h^-|u)} =$$

$$logN(0;0,\sigma^2 I_n) - \frac{1}{2\sigma^2} \left[Tr(K_{nn}) - Tr(Q_{mm}) \right]$$

$$-\frac{1}{2\sigma^2} \langle h^T h - 2\mathcal{H}_{nm} u + u^T \alpha_{nm}^T \alpha_{nm} u \rangle$$

Taking expectations with respect to u, this evaluates to:

$$logN(0;0,\sigma^2 I_n) - \frac{1}{2\sigma^2} \left[Tr(K_{nn}) - Tr(Q_{mm}) \right]$$

$$-\frac{1}{2\sigma^2} \langle Tr \left[\left(\alpha_{nm}^T \alpha_{nm} \right) \left(\Sigma + \mu^T \mu \right) \right] - 2\mathcal{H}_{nm} \mu + Tr(\Sigma_h + u_h u_h^T) \rangle$$

The information above is sufficient to compute the model the bound. The only difference in computational complexity from the mean-field approach is that the Markov dynamics are $O(d)$ for each layer, dimension and data-point. This is the same as the number of variational parameters. To form this bound, we would also have to compute the new kernel statistic $\mathcal{H}_{nm} = \langle h^T K_{nm} \rangle$.

Since $q(h^-)$ and $q(h)$ have diagonal covariances, this new statistic will be $O(nd)$ to compute, which means the computational complexity scales with $O(lnd^2)$, since this term must be evaluated for d kernels K^d_{nm}., l layers and n data-points. The complexity and cost arising from the pseudo-outputs remains the same. Thus:

Compute Time	Variational parameters
O($l(nd^2 + nm^2)$)	$O(lnd^2)$

As such, this bound would be quite tractable for many data-sets.

4.7 Dependencies between Layers $q(u^l|u^{l-1})q(h^l|u^l)$:

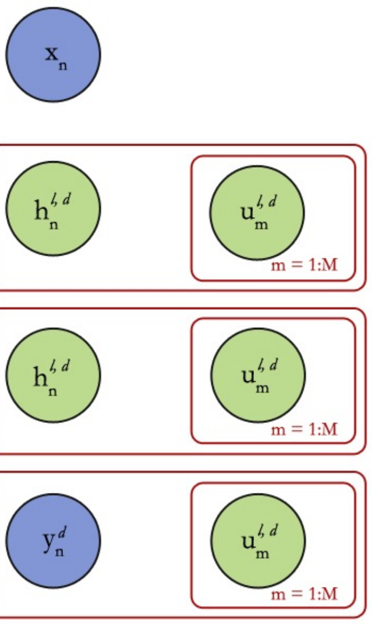


Figure 4.8: Fully Connected Restricted Boltzman Machine

Ideally, we would like to model the dependencies between h and u, whilst also modelling dependencies between layers. However, such a modelling assumption would render it impracticable to find an optimal distribution for $q(u)$. To understand why this is so, consider that in the examples discussed, it has been necessary to compute the expectation $\langle K^{ld}_{nm} \rangle_{q(h|u)}$ to determine the sufficient statistics for the optimal distribution $q(u)$ (at a given layer) . That is to say, $q(u)$ has been of the form $q(u) \propto e^{-u^T f(\langle K^{ld}_{nm} \rangle)u + ..}$ for some linear function f.

If we introduce dependencies between h and u, whilst also modelling dependencies between layers, then h will depend on u, and so$\langle K^{ld}_{nm} \rangle_{q(h|u)}$ will be some non-linear function of u. This means the optimal distribution for $q(u)$ will no longer be Gaussian, since $\langle K^{ld}_{nm}\rangle_{q(h|u)}$ must be independent of u for $q(u)$ to be Gaussian.

As a result of this, we would not be able to compute the expectations $\langle u\rangle$ or $\langle u^T u\rangle$ which were required when computing the optimal lower bounds.

Having said all of this, we may still construct a model where there are dependencies between both u and h variables as well as having dependencies through layers, we just can't optimally integrate out $q(u)$ when determining the bound. In this section, we will explore these modelling assumptions.

4.7.1 Variational Approximation

The variational approximation that we shall use is:

$$q(u,h) = \prod_{l,d} p(f^{ld} \neq u^{ld}|u^{ld}) q(u^{ld}|u^{l-1d}) \prod_n q(h^{ld}|u^{ld})$$

Here we have modelled the dependencies between layers using a Markov transition:

$$q(u^{ld}|u^{l-1d}) = N(u^{ld}|\mathcal{U}^{\mathrm{ld}} u^{l-1d} + \mathcal{V}^{ld}, \mathcal{W}^{ld})$$

$$q(h^{ld}|u^{ld}) = \prod_n N(h^{ld}_n|\mathcal{A}^{\mathrm{ld}}_{\mathrm{n}} u^{ld} + \mathcal{B}^{ld}_n, \mathcal{C}^{ld}_n)$$

4.7.2 Markovian Dynamics

Given these conditional distributions, we can write down the dynamics of the system, which will prove to be useful in later sections. We drop subscripts l, d for ease of notation, and denote the previous layer with a #, (rather than a $-$ sign for improved clarity). Below we note the marginal distribution for different variables which may be calculated by evaluating the Markovian dynamics. We use Roman letters for the marginal distribution of h, and Greek for the marginal distribution of $q(u)$.

$$q(u) = N(u; \mu, \Sigma), \text{ where} \mu = \mathcal{U}\mu^{\#} + \mathcal{V},\ \Sigma = \mathcal{U}\Sigma^{\#}\mathcal{U}^T + \mathcal{W}$$

$$q(h_n) = q(h_n; m_n, S_n), \text{ where } m_n = \mathcal{A}_{\mathrm{n}}\mu^{\#} + \mathcal{B}_n,\ S_n = \mathcal{A}_{\mathrm{n}} S^{\#}_n \mathcal{A}^T_n + \mathcal{C}_n)$$

As the base case of the recursion:

$$m^0_n = 0, \mu^0 = 0, \Sigma^0 = 0$$

4.7.3 The Bound:

Following similar procedures to those described in chapter (2), the bound may be written as:

$$F(q) = \sum_{l,d} \langle \mathcal{E}^{ld} \rangle_{q(h,u)} - \sum_{l,d} \int q(u^{l-1d}) \mathrm{KL}(q(u^{ld}|u^{l-1d})|p(u^{ld})) + \sum_{l,d,n} \int q(u^{ld}) \mathrm{H}(q(h_n^{ld}|u^{ld}))$$

Considering only the terms that depend on l, d, we may drop the subscript for the time being.

$$F(q) = \langle \mathcal{E} \rangle_{q(h,u)} - \int q(u^{\#}) \mathrm{KL}(q(u|u^{\#})|p(u)) + \sum_n \int q(u) \mathrm{H}\left[q(h_n|u)\right] \tag{4.7}$$

4.7.4 Evaluating the Entropy and KL Term

Since the Entropy and KL terms are the least challenging aspects of the bound to compute, we consign the evaluation of these expression to the appendix.

4.7.5 Evaluating the $\mathcal{E}$ Term

To evaluate $\sum_{l,d} \langle \mathcal{E}^{ld} \rangle_{q(h,u)}$, we will consider the part of $\mathcal{E}$ that relates to a single dimension and layer, so: $h = h^{ld}$, $h^{\#} = h^{l-1}$, $u = u^{ld}$, $K_{nm} = K_{nm}^{ld}$. Given this, let $\alpha_{nm} = K_{nm} K_{mm}^{-1}$, $Q_{mm} = K_{mm}^{-1} K_{mn} K_{nm}$ .Given the work from the preceding sections, we may write:

$$\mathcal{E} = logN(0|0, \sigma^2 I_n) - \frac{1}{2\sigma^2}\left[Tr(K_{nn}) - Tr(Q_{mm})\right]$$

$$-\frac{1}{2\sigma^2}\langle (h^T h - 2h^T \alpha_{nm} u + u^T \alpha_{nm}^T \alpha_{nm} u)$$

We now need to take expectations, of $\mathcal{E}$ with respect to $q(h, h^{\#}, u)$. Note that kernel statistics already exist for $Tr(K_{nn}) - Tr(Q_{mm})$ for a wide variety of kernels. What remains to be computed is the second line of the formula above, i.e:

$$-\frac{1}{2\sigma^2}\langle h^T h - 2h^T \alpha_{nm} u + u^T \alpha_{nm}^T \alpha_{nm} u \rangle_{q(h,h^{\#},u)}$$

Computing the $\langle h^T h \rangle$ is simple, as it may be expressed as $Tr[S + mm^T]$, given S and m as determined by the Markovian dynamics for $q(h)$. However, it will also be necessary to compute new kernel statistics for the terms: $\langle 2h^T \alpha_{nm} u \rangle, \langle u^T \alpha_{nm}^T \alpha_{nm} u \rangle$.

4.7.6 Evaluating the Kernel Statistic $\langle 2h^T \alpha_{nm} u \rangle_{q(h^\#,u,h)}$:

The general strategy to determine $\langle 2h^T \alpha_{nm} u \rangle_{q(h^\#,u,h)}$ will be to follow the Markovian dynamics by rewriting $h^T \alpha_{nm}^T u$ in terms of only the variable $h^\#$, thereby eliminating u and h from the expectation. Having done this, we can then compute kernel statistic by considering only a single random variable $h^\#$. We will need to use the following result about the conditional distributions within our Markov model:

The conditional distribution of $q(u, h|h^\#)$ is a conditional Gaussian:

$$q(\begin{bmatrix} u|h^\# \\ h|h^\# \end{bmatrix}) \sim N(\begin{bmatrix} \mu_u - \Sigma_{u,h^\#}\Sigma_{h^\#}^{-1}(h^\# - \mu_{h^\#}) \\ \mu_h - \Sigma_{h,h^\#}\Sigma_{h^\#}^{-1}(h^\# - \mu_{h^\#}) \end{bmatrix}, \begin{bmatrix} \cdots & \Sigma_{u,h^\#} - \Sigma_{u,h^-} \cdot \Sigma_{h^\#}\Sigma_{h^\#,h} \\ \iddots & \vdots \end{bmatrix})$$

Terms such as $\Sigma_{u,h^\#}$, a may be determined by considering the Markovian dynamics, for instance:

$$\Sigma_{h^\#,h^\#} := S^\# \text{ as previously defined}$$

$$\Sigma_{h,h^\#} = cov(\mathcal{A}^\# h^\#, h^\#) = \mathcal{A}^\# S^\#$$

$$\Sigma_{u,h^\#} = cov(u, h^\#) = cov(\mathcal{U}^\# h, h^\#) = cov(\mathcal{U}\mathcal{A}^\# h^\#, h^\#) = \mathcal{U}\mathcal{A}^\# S^\#$$

Figure 4.9: **Conditional Gaussian of** $q(u, h|h^\#)$

Using the trace trick, we may write:

$$\langle 2h^T \alpha_{nm} u \rangle_{q(h^\#,u,h)} = \langle 2Tr[a_{nm} h u^T] \rangle_{q(h^\#,u,h)}$$

We can compute the expectations of hu^T, given that we have the conditional distribution for $q(u, h|h^\#)$:

$$\langle h^T \alpha_{nm} u \rangle_{q(h^\#,u,h)} = \langle Tr[a_{nm} \left[\Sigma_{u,h|h^\#} - \mu_{h|h^\#}\mu_{u|h^\#}^T\right] \rangle_{q(h^\#)}$$

This evaluates to:

$$Tr\langle a_{nm} \left[\Sigma_{u,h|h^\#} - \left[\mu_h - \Sigma_{h,h^\#}\Sigma_{h^\#}^{-1}(h^\# - \mu_{h^\#})\right] \left[\mu_u - \Sigma_{u,h^\#}\Sigma_{h^\#}^{-1}(h^\# - \mu_{h^\#})\right]^T\right] \rangle_{q(h^\#)}$$

We now use the following shorthand:

A=$(\mu_h + \Sigma_{h,h^\#}\Sigma_{h^\#}^{-1}\mu_{h^\#})$, B=$(\mu_u + \Sigma_{u,h^\#}\Sigma_{h^\#}^{-1}\mu_{h^\#})$,C=$\Sigma_{u,h^\#}\Sigma_{h^\#}^{-1}$, D=$\Sigma_{h,h^\#}\Sigma_{h^\#}^{-1}$, then

This expression can be factorised in the following manner:

$$Tr\left\langle a_{nm}\left(\Sigma_{u,h|h^\#} - AB^T + A(Ch^\#)^T + Dh^\# B + Dh^\#(Ch^\#)^T\right)\right\rangle$$

This expression is tractable, but we will need to compute the following kernel statistics:

$$\langle a_{nm} \rangle, \ \langle a_{nm} D h^{\#} \rangle, \ \langle a_{nm} A C h^{\#T} \rangle, \ \langle a_{nm} D h^{\#} h^{\#T} C^T \rangle$$

For constants A, B, C, D. These kernel statistics boil down to convolutions of Gaussians, and so are analytic. A question arises as to whether they would be tractable to compute, given that we woud want to compute these statistics in ideally O(d) for high-dimensional data. A second order kernel statistic (for instance ones with $h^{\#T} h$ terms) will take at least $O(d^2)$ time to compute, if no low-rank approximations are used. In a later section, we will show how to compute some of these kernel statistics.

4.7.7 Evaluating the Kernel Statistic $\langle u^T \alpha_{nm}^T \alpha_{nm} u \rangle_{q(h^{\#}, u, h)}$:

The general strategy to determine this expectation will be the same as given in the preceding section. We will rewrite $u^T \alpha_{nm}^T \alpha_{nm} u$ in terms of $h^{\#}$, eliminating u, and then to finally compute the expectation by only considering only the r.v. $h^{\#}$.

Using the trace trick, we may evaluate $\langle u^T \alpha_{nm}^T \alpha_{nm} u \rangle_{q(u|h^{\#})}$ as:

$$\langle u^T \alpha_{mn} \alpha_{nm} u \rangle_{q(u|h^{\#})} = Tr \left[(\alpha_{nm}^T \alpha_{mn})(\Sigma_{u|h^{\#}} + \mu_{u|h^{\#}} \mu_{u|h^{\#}}^T) \right]$$

Note that $\mu_{u|h^{\#}} \mu_{u|h^{\#}}^T$ may be written as:

$$\left[\mu_u - \Sigma_{u,h^{\#}} \Sigma_{h^{\#}}^{-1} (h^{\#} - \mu_{h^{\#}}) \right] \left[\mu_u - \Sigma_{u,h^{\#}} \Sigma_{h^{\#}}^{-1} (h^{\#} - \mu_{h^{\#}}) \right]^T$$

We must separate terms that depend on $h^{\#}$. Much like before, this evaluates to:

$$\text{A}=(\mu_u + \Sigma_{u,h^{\#}} \Sigma_{h^{\#}}^{-1} \mu_{h^{\#}}),\text{B}=\Sigma_{u,h^{\#}} \Sigma_{h^{\#}}^{-1}$$

$$Tr \left\langle \alpha_{nm}^T \alpha_{nm} \left(\Sigma_{u,h|h^{\#}} - AA^T + 2A(Bh^{\#})^T + (Bh^{\#})(Bh^{\#})^T \right) \right\rangle$$

This expression is tractable, but we will need to compute the following kernel statistics:

$$\langle \alpha_{nm}^T \alpha_{nm} \rangle, \ \langle \alpha_{nm} 2A(Bh^{\#})^T \rangle, \ \langle \alpha_{nm} \alpha_{mn} (Bh^{\#})(Bh^{\#})^T \rangle$$

Computing these kernel statistics will ultimately come down to convolving Gaussians, and so will be tractable. However, the same comments about dimensionality apply to these kernel expectations as in the preceding section. They will be tractable to compute for low dimensional data, but computing higher order statistics of Gaussian distributions becomes more difficult as the dimensionality of the data grows in size.

4.8 Computing New Kernel Statistics

4.8.1 Introduction

In this section, we will compute some of the new kernel statistics mentioned in the preceding section to give a more concrete feel for the steps that need to be undertaken and to indeed show that these new kernel statistics are tractable. In this section we will make use of the natural parameter form of a Gaussian. Before we begin, we will need to establish our notation.

General Notation:

- Given $\Phi(\eta, \Lambda)$, which is a Gaussian function with natural mean η and precision Λ, we define shorthands for the normalising constant :

$$\zeta := cons(\eta, \Lambda) := \frac{1}{2}(d\log(2\pi) - log|\Lambda| + \eta^T \Lambda \eta)$$

DGP Specific Notation for evaluating K_{nm}

The notation to be discussed pertains to kernel matrices K_{nm}

- Take z_m as input location to u_m which appears in the kernel K_{nm}
- Take $\zeta_q, \eta_q, \Lambda_q$ as the partition function, the natural mean and the precision for the Gaussian $q(h_n^{\#})$, which we use to take expectations over the r.v. K_{nm}.
- We will now drop subscripts n for ease of notation. In this section, K_m will refer to a single element of the kernel matrix K_{nm}^{ld}, which may be defined as the following scaled Gaussian Φ (in natural parameters):

$$K_m = \sigma^2 \Phi(x_n; \eta_{km}, \Lambda_K) exp(\zeta_{km})$$

  In this setting:

  - The mean is centered on the pseudo-input z_m, i.e: $\eta_m = \Lambda_K z_m$
  - The precision matrix Λ_K is the parameter for the kernel (similar to the length-scales matrix but not necessarily diagonal), and therefore the same for all kernel entries K_m.
  - Normalising constant is by definition:

$$\zeta_{km} = cons(\eta_m, \Lambda_k) = \frac{1}{2}(dlog(2\pi) - log|\Lambda_k| + \eta_{km}^T \Lambda_k \eta_{km})$$

Product of Natural Gaussians:

$$\Phi(x, \eta_1, \Lambda_1)\Phi(x, \eta_2, \Lambda_2) \exp\left[\zeta(\eta_1, \Lambda_1) + \zeta(\eta_2, \Lambda_2)\right] =$$

$$\Phi(x, \zeta_1 + \zeta_2, \Lambda_1 + \Lambda_2) \exp\left[\zeta(\eta_1 + \eta_2, \Lambda_1 + \Lambda_2)\right]$$

Figure 4.10: Product of Natural Gaussians

4.8.2 Simple Convolution

Given this setup, we may write the expectation over entries of the kernel as convolutions of Gaussians in their natural parameter forms, i.e:

$$\langle K_m \rangle_{q(h^{\#})} = \int_{h^{\#}} \sigma^2 \Phi(h^{\#}; \eta_{km}, \Lambda_k) \Phi(h^{\#}; \zeta_q, \eta_q, \Lambda_q) exp(\zeta_{km})$$

Using the product of Gaussians result, above, we may rewrite this convolution as a scaled Gaussian. We have put this result in a box below because of its importance. The result may be used to derive a number of different kernel statistics.

Result 1:

$$K_m q(h^{\#}) = c_m \cdot p(h^{\#}) = c\mathcal{N}(h^{\#}; \eta_{km} + \eta^q, \Lambda_k + \Lambda_q)$$

where:

$$c_m = \sigma^2 e^{[-\zeta(\eta_{km}+\eta^q, \Lambda_k+\Lambda_q)+\zeta(\eta_q,\Lambda_q)-\zeta(\eta_{km},\Lambda_k)]}$$

Figure 4.11: Simplifying the integrand of the kernel statistic

Given this, the kernel expectation, which we wished to compute, neatly evaluates to:

$$\langle K_m \rangle_{q(h^{\#})} = \int_{h^{\#}} c_m \cdot p_m(h^{\#}) = c_m$$

In the case of diagonal covariance matrices and an ARD Kernel, this computation is $O(d)$, but otherwise it is $O(d^3)$. Later we will present an idea which will enable us to use non-diagonal covariance and SE-kernels to compute these new kernel expectations in $O(d)$.

4.8.3 Evaluating the new Kernel Statistics:

With the new framework and notation in place from the preceding section, we may go on to evaluate more complicated kernel statistics with relative ease. As an example, recall from the preceding section, that it was necessary to compute $\langle a_{nm} D h^{\#} \rangle$, $\langle a_{nm} D h^{\#} h^{\#T} C^T \rangle$. Factorising by n, for the first of these expectations, it comes down to being able to compute:

$$\chi = \langle \underset{1\times m}{K_m} \underset{m\times d}{D} \underset{d\times 1}{h^{\#}} \rangle_{q(h^{\#})}$$

$$\chi = \sum_m \underset{1\times d}{D_m} \langle \underset{1\times 1}{K_m} \underset{d\times 1}{h^{\#}} \rangle_{q(h^{\#})}$$

Thus we need only determine:

$$\chi = \langle \underset{1\times 1}{K_m} \underset{d\times 1}{h^{\#}} \rangle_{q(h^{\#})}$$

Given the work from the preceding section, this evaluates to:

$$\chi = \langle c_m \underset{d\times 1}{h^{\#}} \rangle_{p(h^{\#})} = c_m \mu_{p_m}$$

4.8.3.1 Evaluating $\langle Tr[a_{nm} D h^{\#} h^{\#T} C^T] \rangle$

As for the second of these kernel statistics:

$$\langle Tr[a_{nm} D h^{\#} h^{\#T} C^T] \rangle_{q(h^{\#})}$$

To compute this, factorising by n, we need only compute:

$$= \langle Tr[\sum_m \underset{1\times 1}{K_m} \underbrace{K_{mm}^{-1} D}_{1\times d} \underset{d\times 1}{h^{\#}} \underset{1\times d}{h^{\#T}} C^T] \rangle_{q(h^{\#})}$$

$$= Tr[\underbrace{\sum_m K_{mm}^{-1} D}_{1\times d} \left\langle K_m \underset{d\times 1}{h^{\#}} \underset{1\times d}{h^{\#T}} \right\rangle_{q(h^{\#})} C^T]$$

But given the work from the previous section, we know that this evaluates to:

$$\langle \underset{d\times 1}{h^{\#}} \underset{1\times d}{h^{\#T}} c_m \rangle_{p(h)} = c_m (\Sigma_{p_m(h^{\#})} - \mu_{p_m} \mu_{p_m}^T)$$

4.8.4 Computing Other Kernel Statistics

The other kernel statistics involve $a_{mn} a_{nm}$ terms and so, $K_{mm}^{-1} \sum_n K_{mn} K_{nm} K_{mm}^{-1}$ terms. The key thing to note is that this sum factors over n and so once again, we can simplify our work by focusing on statistics for a single data-point, i.e:

$$\langle \underset{d\times 1}{h^{\#}} \underset{1\times d}{h^{\#T}} K_{mn} K_{nm} \rangle_{q(h^{\#})} \text{ or } \langle \underset{d\times 1}{h^{\#}} K_{mn} K_{nm} \rangle_{q(h^{\#})} \text{ or } \langle K_{mn} K_{nm} \rangle_{q(h^{\#})}$$

Without delving to deep into the theory, as before we can take the product $K_{mn} K_{nm} q(h^{\#})$ and form a new scaled Gaussian distribution in $h^{\#}$. This will be just as tractable to compute as previous convolutions. At a first glance, it seems the extra cost of computing these convolutions ought to be more expensive to compute than the previous kernel statistics, an entry being computable in $O(d)$ for ARD kernels.

4.9 Extending Inference to Higher Dimensions:

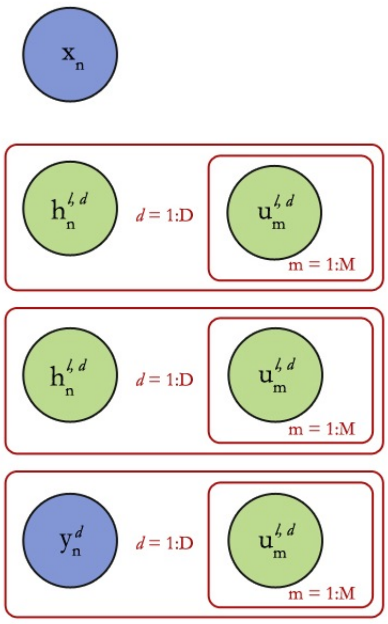


Figure 4.12: Fully Connected Restricted Boltzman Machine

It is possible to extend the method discussed in the preceding section to higher dimensions. We now model the dynamics in a manner that is no longer factored by dimension, i.e.:

$$q(u^l|u^{l-1}) = N(\mathcal{U}^{\mathrm{l}}u^{l-1} + \mathcal{V}^l, \mathcal{W}^l)$$

$$q(h^l|u^l) = \prod_n N(h_n^l|\mathcal{A}_{\mathrm{n}}^{\mathrm{l}}u^l + \mathcal{B}_n^l, \mathcal{C}_n^l)$$

Many of the comments and proofs from the preceding section would carry through as before. The bound, now factored only by layer would become:

$$F(q) = \sum_{l,d}\langle\mathcal{E}^{ld}\rangle_{q(h,u)} - \sum_{l,d}\int q(u^{l-1})\mathrm{KL}(q(u^l|u^{l-1})|p(u^{ld})) + \sum_{l,d,n}\int q(u^l)\mathrm{H}(q(h_n^l|u^l))$$

The KL and Entropy terms would be changed only very slightly to compute in this higher dimensional setting. For example, the KL term would now become:

$$\frac{1}{2}\left[\log\frac{|K|}{|S_n|} - D + Tr(K^{-1}S_n) + Tr[\mathcal{B}_{\mathrm{n}}^{\mathrm{T}}\mathrm{K}^{-1}\mathcal{B}_{\mathrm{n}}(\Sigma^{\#} + \mu^{\#T}\mu^{\#})] + m_n{}^T K^{-1}m_n) + 2m_nK^{-1}\mathcal{B}_{\mathrm{n}}\mu^{\#})\right]$$

where K is a md×md block diagonal:

$$\begin{pmatrix} k_{mm}^{l1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & K_{mm}^{lD} \end{pmatrix}$$

With regards to computing the expectation $\sum_{l,d}\langle\mathcal{E}^{ld}\rangle_{q(h,u)}$. This would be almost exactly the same as in the preceding section, the only difference being that the covariance matrix and mean of the conditional distribution $q(u,h|h^{\#})$ used to compute the expectation would be a factor of d larger. But given this, we may compute kernel statistics for $\langle 2h^T\alpha_{nm}u\rangle$, and $\langle u^T\alpha_{nm}^T\alpha_{nm}u\rangle$, following the Markovian dynamics in precisely the same manner. The difference is that these terms will be a factor of$O(d)$ or $O(d^2)$ more expensive to compute.

Finally, we defer a discussion about reducing the computational expense of methods that model dependencies through dimension to the appendix. The method will aim to scale in $O(d^2)$.

4.10 Concluding Remarks on Which Approximation to use

We have covered a great many different variational approximations in this chapter. The key questions is which variational distribution deserves our best attention?

In my opinion, it would be the two simplest. These are $q(h^l|h^{l-1})$ and also $q(h^l|u^{l-1})$. These variational approximations require little more machinery than is already required for the mean-field bound. They do not significantly change the compute time of the bound, and only use a relatively small multiple more variational parameters. In fact, these extra variational parameters may not even be so much of a worry since sophisticated parameter tying techniques could be used to reduce their number, which we have not discussed here.

Of theses two distributions, I would expect the most significant modelling improvements to be had from the variational distribution for $q(h^l|u^{l-1})$. This is because from the model's point of view, conditioned on the $u's$, the distribution of $q(h)$ takes the form:

$$(1)\ mean(h|u) = K_{nm}K_{mm}^{-1}u$$

$$(2)\ cov(h|u) = K_{nn} - K_{nm}K_{mm}^{-1}K_{mn} + \sigma^2 I_n$$

This distribution is somewhat mimicked by, our variational distribution since:

$$mean[q(h|u)] = \mathcal{T}u$$

Here we see the u term appear in the conditional of our new variational distribution $q(h|u)$. This shows that the modelling assumption emulates the model.

What I had further noticed from my practical work, was that although the mean of the pseudo-points often settled down to a stable value, there was a much greater propensity for the means of $q(h^l)$ to jostle around, seeming to tune themselves to noise around the function. Perhaps by introducing more dependencies between the $u's$ and the $h's$ we would see more stability. Only further empirical work would determine this.

With regards to the variational distribution $q(h^l|h^{l-1})$ that models the dependencies between h through layers, we may see benefits from the distribution since there are clearly dependencies between a function's input values and its output values.

Unlike these two distributions, the other variational distributions would have some notable disadvantages, for instance: when modelling more dimensions, the parameters tend to scale in $O(d^3)$; when modelling dependencies between h and u and also layers, we can then no longer find the optimal distributions for $q(u)$.

Ultimately we hope that this exposition provides some insights and ideas to others who wish to challenge the modelling assumptions in a DGP. If we were to pick just one new variational approximation to take forward, it would be the one that models the dependencies of h on u, i.e. $q(h|u)$.

Chapter 5

Concluding Remarks

In what follows we make a brief statement of the contributions from this thesis, before describing future research areas of research with DGPs.

5.0.1 Summary of Contributions

Our thesis began, in chapter 2, with an introductory description of sparse GPs, which aimed to stress the intuitions behind the method developed by Titsias 09 rather than the mathematics. Having done this, we showed how the method may then be directly applied to DGPs, thereby providing a grounding for later theoretical research into modelling the dependencies in DGPs. Ultimately, we hope that this first section will aid researchers who hope to challenge the variational modelling assumptions used within the DGP framework.

In chapter 3, we went on to describe our implementation of a fully-functional DGP in Tensorflow, which was tested on a number of toy problems, such as the step function considered by Damianou [2015] and also a sinusoid. We hope the implementation will provide a clear starting point for practitioners who want to bring the full compute power of GPUs to force on the DGP model.

Along the way, we also produced implementations of sparse GPs which were optimised using both a natural gradient based scheme and by optimally integrating out the distribution over pseudo-points $q(u)$. In the end , we concluded that Titsias' approach worked best. We then further analysed different gradient-based optimisation schemes, discovering that conjugate gradients outperformed the standard gradient-based optimsation methods used in Tensorflow when training DGPs.

With regards to theoretical contributions, we produced a 30 page analysis into a range of different variational free energy approximations to the log-likelihood, based on different variational modelling assumptions. We investigated how to model dependencies between layers, between dimensions, and between hidden layers and inducing points- providing a novel contribution to this particular area of modelling. I would here like to thank Richard Turner for the ideas and helpful suggestions with regard to modeling these dependencies. Within this section, we often used Titsias' method to optimally parameterise $q(u)$, motivated in part by the practical experience gained from chapter 3.

5.0.2 Future work in DGPs

Building from the research developed in this thesis, we would like to see the simplest of the new variational free-energy bounds implemented. If time had permitted, I would have developed an implementation that modeled the dependencies between hidden variables h and inducing outputs u. Not only would this bound be the simplest to compute, but due to the strong dependencies between u and h[1] has a great deal of potential to improve upon the mean-field variational modelling assumptions.

If time had permitted, we would also have liked to test how the Tensorflow DGP implementation performs on more interesting test-cases, for instance multi-dimensional regression problems. However, the model has been programmed in a way that would immediately allow testing in a multi-dimensional setting. Only slight changes need be introduced to increase the number of layers in the DGP architecture.

[1]Titsias [2009] describes u as a sufficient statistic for f, which is closely related to h

References

[1] Andreas Damianou. Deep gaussian processes and variational propagation of uncertainty. *PhD Thesis, University of Sheffield*, 2015.

[2] Andreas Damianou and Neil Lawrence. Deep Gaussian processes. In C. Carvalho and P. Ravikumar, editors, *Proceedings of the Sixteenth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, AISTATS '13, pages 207–215. JMLR W&CP 31, 2013.

[3] Gal et al. Variational inference in sparse gaussian process regression and latent variable models â a gentle tutorial. *arxiv*.

[4] James Hensman and Neil D Lawrence. Nested variational compression in deep Gaussian processes. *arXiv preprint arXiv:1412.1370*, 2014.

[5] James Hensman, Nicolo Fusi, and Neil D Lawrence. Gaussian processes for big data. In *Conference on Uncertainty in Artificial Intellegence*, pages 282–290. auai.org, 2013.

[6] Antti Honkela, Matti Tornio, Tapani Raiko, and Juha Karhunen. Natural conjugate gradient in variational inference. In *Neural Information Processing, 14th International Conference, ICONIP 2007, Kitakyushu, Japan, November 13-16, 2007, Revised Selected Papers, Part II*, pages 305–314, 2007. doi: 10.1007/978-3-540-69162-4_32. URL http://dx.doi.org/10.1007/978-3-540-69162-4_32.

[7] Alexander G de G Matthews, James Hensman, Richard E Turner, and Zoubin Ghahramani. On sparse variational methods and the Kullback-Leibler divergence between stochastic processes. *Proceedings of the Nineteenth International Conference on Artificial Intelligence and Statistics*, 2016.

[8] Yingzhen Li Jose Miguel Hernandez-Lobato Richard E. Turner Thang D. Bui, Daniel Hernandez-Lobato. Deep gaussian processes for regression using approximate expectation propagation. *arxiv*.

[9] Titsias. Variational learning of inducing variables in sparse gaussian processes. *JMLR*.

[10] Michalis K. Titsias and Neil D. Lawrence. Bayesian gaussian process latent variable model. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pages 844–851, 2010. URL http://www.jmlr.org/proceedings/papers/v9/titsias10a.html.

Appendix A

Theoretical Appendix

A.1 Evaluating the Likelihood $\mathcal{E}(y|x,u) = \langle log[p(y|f \neq u, x)\rangle_{p(f \neq u|u)}$

We can first unpack $p(y|f \neq u, x)$. Recall that y is simply $f \neq u|x$ but with added noise. The expression $p(y|f \neq u, x)$ may therefore be unpacked as:

$$\langle log[p(y|f \neq u, x)\rangle_{p(f \neq u|u)} = \langle \log N(y; f \neq u; \sigma^2 I)\rangle_{p(f \neq u|u)}$$

Expanding this term and piping the expectations inwards, we obtain:

$$= logN(0; 0, \sigma^2 I_n) - \frac{1}{2\sigma^2}[y^T y - 2y^T \langle f \neq u\rangle + \langle (f \neq u)^T (f \neq u)\rangle]$$

Note that we have dropped the $p(f \neq u|u)$ subscripts on expectations for clarity. Completing the square in $(y - \langle f \neq u\rangle)^T$, we form (†) below:

$$= logN(0; 0, \sigma^2 I_n) - \frac{1}{2\sigma^2}[(y - \langle f \neq u\rangle)^T (y - \langle f \neq u\rangle)]$$

$$-\frac{1}{2\sigma^2} Tr[cov(f \neq u|u)]$$

To evaluate (†) , we further require the following results:

$$\langle f \neq u\rangle_{p(f \neq u|u)} = K_{nm} K_{mm}^{-1} u \tag{A.1}$$

$$cov(f \neq u|u) = K_{nn} - K_{nm} K_{mm}^{-1} K_{mn} \tag{A.2}$$

With this is mind, we the preceding results in mind we may express $\mathcal{E}(y|x,u)$ as:

$$\mathcal{E}(y|x,u) = logN(y; K_{nm} K_{mm}^{-1} u, \sigma^2 I) - \frac{1}{2\sigma^2}\mathrm{Tr}[K_{nn} - K_{nm} K_{mm}^{-1} K_{mn}] \tag{A.3}$$

A.2 Reversing Jensen's inequality (Titsias 09, optimal $F(q)$)

Consider once again the expression for the lower bound obtained from (1) :

$$F(q) = \left\langle \mathcal{E}(y|x,u) - log\frac{q(u)}{p(u)} \right\rangle_{q(u)} \tag{A.4}$$

Since no constraints were imposed on q, while optimising, we may reverse Jensen's inequality to obtain an maximised variational lower bound with respect to the distribution q. This leads to the result:

$$F(q) = \left\langle \mathcal{E}(y|x,u) - log\frac{q(u)}{p(u)} \right\rangle_{q(u)} = \left\langle \log\left[e^{\mathcal{E}(y|x,u)} \frac{p(u)}{q(u)} \right] \right\rangle_{q(u)} \leq \log\langle e^{\mathcal{E}(y|x,u)} \rangle_{p(u)}$$

Given the expression computed for $\mathcal{E}(y|x,u)$, it can be seen that $\log\langle e^{\mathcal{E}(y|x,u)}\rangle_{q(u)}$ is:

$$= \log\left\langle N(y; K_{nm}K_{mm}^{-1}u, \sigma^2 I) \right\rangle_{p(u)} - \frac{1}{2\sigma^2} Tr[K_{nn} - K_{nm}K_{mm}^{-1}K_{mn}]$$

Note that the term $\tilde{p}(y|u) = N(y; K_{nm}K_{mm}^{-1}u, \sigma^2 I)$ models the probability of y as a linear transform of u, where: $y = K_{nm}K_{mm}^{-1}u + \epsilon$ with $\epsilon \sim \sigma^2 I$. And given the result A.1 about linear transformations of Gaussian variables, the 'marginal' distribution can be determined as below, where $Q_{nn} = K_{nm}K_{mm}^{-1}K_{mn}$.

$$\left\langle N(y; K_{nm}K_{mm}^{-1}u, \sigma^2 I) \right\rangle_{p(u)} = \tilde{p}(y) = \mathrm{N}(0; \sigma^2 I + Q_{nn})$$

For independent variables x and ϵ, where:
$y = Ax + b$, where $\epsilon \sim N(0, \Sigma_\epsilon)$ and $x \sim N(\mu_x, \Sigma_x)$
We know: $y \sim N(A\mu_x,\ A\Sigma_x A^T + \Sigma_\epsilon)$

Figure A.1: Linear Transform of Gaussian Distributed Variables

A.3 Why DGPs need Pseudo-Points

Once again, our aim is to find an appropriate term for the log-likelihood of our model and to then optimise this with respect to the model parameters. The log-likelihood of a deep GP may be specified as

$$\log p(y|x) = \sum_{l=0}^{L} \log \int p(h^{l+1}|h^l)$$

Where we have considered $h^{l+1} = y$ and $h^0 = x$. We would like to use a standard mean-field variational approximation to model the probability distribution within hidden layers, since marginalising through the possibilities within all the hidden layers is intractable. This variational distribution takes the form: $q(h) = \prod_l h_l$. Doing so, we would obtain the following bound:

$$\log p(y|x) \geq \sum_l log\langle p(h^{l+1}|h^l)\rangle_{q(h^{l+1})q(h^l)} + \sum_l \mathrm{H}(q(h^l))$$

However, as has been remarked by Damianou [2015], this bound remains intractable. The main reason for this is that to compute the terms $log\langle p(h^{l+1}|h^l)\rangle_{q(h^{l+1})q(h)}$, one must compute $log\langle N(h^{l+1}|0, K_{h^l h^l} + \sigma_l^2 I)\rangle_{q(h^{l+1})q(h^l)}$, since each layer is modelled as the noisy outputs from a Gaussian process. Evaluating this expression is non-trivial since it requires us to compute the expectation of $\langle K^{-1}_{h^l h^l}\rangle_{q(h^l)}$, which pipes a Gaussian distribution through a highly non-linear kernel function.

However, this issue can be remedied through the use of inducing points. So here we come to a second crucial aspect of the inducing points, not only do they make it possible to approximate log-likelihood terms when faced with big-data, they also make inference through the layers of a deep GP tractable. The reason why this is so, is due to the fact that input locations of the inducing points are considered fixed, and so when we use the pseudo-point approximation to $\langle K^{-1}_{h^l h^l}\rangle_{q(h^l)}$ we end up with a term that more closely resembles $\langle K_{h^l m} K^{-1}_{mm} K_{m,h^l}\rangle_{q(h^l)}$, which (being the convolution of two Gaussians) is tractable.

A.4 Markov Modelling Result 1

Given that $h = \mathcal{T}u + \mathcal{M} + \epsilon$, where $\epsilon \sim N(0, \mathcal{S})$
We may write:

$$\langle h^T h\rangle_{q(h|u)} = \langle(\mathcal{T}u + \mathcal{M} + \epsilon)^T(\mathcal{T}u + \mathcal{M} + \epsilon)\rangle_{N(\epsilon;0,S)} = u^T\mathcal{T}^T\mathcal{T}u + 2u^T\mathcal{T}^T\mathcal{M} + Tr[\mathcal{S}]$$

Figure A.2: Markov Expectation Result (1)

A.5 Evaluating the Entropy and KL for the distribution: $q(h^l|u^l)q(u^l|u^{l-1})$

A.5.1 The Entropy:

$$(\dagger) = \sum_n \int q(u)\mathbf{H}\left[q(h_n|u)\right]$$

In our case marginalising through $q(u)$ has no effect on the entropy.

Hence, $(\dagger)$ evaluates to:

$$(\dagger) = \sum_n \left[\frac{D}{2}(1 + \ln(2\pi)) + \frac{1}{2}\ln|\boldsymbol{S_n}|\right]$$

$$\boxed{\begin{gathered}\mathrm{KL}[p_1|p_2] = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - D + Tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right] \\ \mathrm{H(p)}{=}\frac{D}{2}(1+\ln(2\pi)) + \frac{1}{2}\ln|\mathbf{\Sigma}|\end{gathered}}$$

Figure A.3: KL Divergence and Entropy Formulas

A.5.2 Evaluating the KL: $q(h^l|u^l)q(u^l|u^{l-1})$

Given the formula for the KL divergence A.5.1, we have:

$$\int_{u^{\#}} q(u^{\#})\frac{1}{2}\left[\log\frac{|K_{mm}|}{|S_n|} - D + Tr(K_{mm}^{-1}S_n) + (\mathcal{B}_\mathrm{n}u^{\#} + m_n)^T K_{mm}^{-1}(\mathcal{B}_\mathrm{n}u^{\#} + m_n))\right]$$

Using the trace trick, and taking expectations wrt. $q(u^{\#})$, we have:

$$\frac{1}{2}\left[\log\frac{|K_{mm}|}{|S_n|} - D + Tr(K_{mm}^{-1}S_n) + Tr[\mathcal{B}_\mathrm{n}^\mathrm{T}\mathrm{K}_\mathrm{mm}^{-1}\mathcal{B}_\mathrm{n}(\Sigma^{\#} + \mu^{\#T}\mu^{\#})] + {m_n}^T K_{mm}^{-1}m_n) + 2m_nK_{mm}^{-1}\mathcal{B}_\mathrm{n}\mu^{\#})\right]$$

A.6 Reducing the Computational Expense of Variational Modelling through Dimensions (Section 4.9):

If we were to reduce the computational expense of the higher-dimensional method, this could be done by fixing a basis for the Markovian dynamics and re-parameterising the variational parameters with low rank matrices. So originally, we had:

$$q(u^l|u^{l-1}) = N(\mathcal{U}^\mathrm{l}u^{l-1} + \mathcal{V}^l, \mathcal{W}^l)$$

$$q(h^l|u^l) = \prod_n N(h_n^l|\mathcal{A}_\mathrm{n}^\mathrm{l}u^l + \mathcal{B}_n^l, \mathcal{C}_n^l)$$

In our new system, we would model the transitions with low rank $r\times r$ matrices:

$$\mathcal{U}^\mathrm{l} = \underbrace{M}_{md\times r}\underbrace{N_1^l}_{r\times r}\underbrace{O}_{r\times md},\ \mathcal{W}^l = \underbrace{D^l}_{md\times md} + \underbrace{M}_{md\times r}\underbrace{N_2^l}_{r\times r}\underbrace{O}_{r\times md}$$

$$\mathcal{A}_\mathrm{n}^\mathrm{l} = \underbrace{Q}_{\mathrm{d\times r}}\underbrace{\overset{\mathrm{l}}{R}}_{\mathrm{r\times r}}\underbrace{S}_{\mathrm{r\times d}}, \mathcal{C}^l = \underbrace{D^l}_{d\times d} + \underbrace{Q}_{d\times r}\underbrace{R_2^l}_{r\times r}\underbrace{S}_{r\times d}$$

In this setting, computing $q(u)$ given the distribution for $q(u^{\#})$ takes only $O(mdr^2)$ rather than an $O(m^2d^2)$. The trick we use to do this will be that we will never explicitly evaluate a covariance or mean in full as that would cost $O(d^2)$ at least, but instead use the basis vectors to form an implicit representation and just compute the new $r\times r$ low ran representations.

This ultimately gives us a nice trade off between modelling power and speed.

A.6.1 Low rank Kernel Statistics

Within this low dimensional setting, the final question to emerge would be the cost of computing the new kernel statistics, such as $\langle \underset{d\times 1}{h^{\#}} \underset{1\times d}{h^{\#T}} K_{mn} K_{nm} \rangle_{q(h^{\#})}$ or $\langle \underset{d\times 1}{h^{\#}} K_{mn} K_{nm} \rangle_{q(h^{\#})}$ or $\langle K_{mn} K_{nm} \rangle_{q(h^{\#})}$. We may break down the original statistics to the following:

- $c_m = \sigma^2 e^{[-\zeta(\eta_{km}+\eta^q, \Lambda_k+\Lambda_q)+\zeta(\eta_q,\Lambda_q)-\zeta(\eta_{km},\Lambda_k)]}$
- $\langle \underset{d\times 1}{h^{\#}} \underset{1\times d}{h^{\#T}} \rangle_{p^m(h^{\#})}$
- $\langle \underset{d\times 1}{h^{\#}} \rangle_{p^m(h^{\#})}$

Well, it will turn out that all of these expectations may be computed in $O(d)$. Rather than pour over the algebra, we shall provide an intuition as to why this must be so (with SE kernels) if the precision from the Kernel Λ_k is also low rank and uses the same basis as the vectors $\mathcal{A}^l_n = \underbrace{Q}_{d\times r} \underbrace{\overset{l}{R}}_{r\times r} \underbrace{S}_{r\times d}$, from the Markov transition of $q(h^{\#}_n | h^{\#\#}_n)$. Ultimately, this means we must be able to write $\Lambda_k = \underbrace{Q}_{d\times r} \underset{r\times r}{\lambda^k} \underbrace{S}_{r\times d}$. Given this, when undertaking our calculations to determine the lower bound we will only have three types of objects. Diagonal matrices, $(d \times 1)$ vectors, and matrices that are never explicitly evaluated but stored using their low rank representations: $(d \times r)(r \times r)(r \times d)$, crucially with the same basis. Given this, it can be shown that all mathematical operations we wish to define: inversion, determinants, addition, multiplication etc. may be performed in $O(dr^2)$ and will only ever produce other such objects. As such, computing a kernel expectation will be tractable in $O(d)$.

Appendix B

Code Appendix

B.1 Sparse GP Regression Implementation:

```python
# Helper Routines for Numerical Computation and Actual Bound in Fbound
import numpy as np
import scipy as sp
import matplotlib.pylab as plt
import tensorflow as tf

S=None
seed=1
tol=1e-2

#MAT MUL
def Mul(*arg):
    if len(arg)==1:
        return arg[0]
    else:
        return tf.matmul(arg[0],Mul(*arg[1:]))


def get_dim(X,index):
     shape=X.get_shape()
     return int(shape[index])

#takes cholesky factor as argument
def safe_chol(A,RHS):
    conditioned=condition((A+tf.transpose(A))/2)
    chol=tf.cholesky(conditioned)
    return tf.cholesky_solve(chol,RHS)

def isclose(X):
    Z= abs_diff(X,X)+np.eye(get_dim(X,0),get_dim(X,0))
    return S.run(tf.to_float(tf.reduce_min(Z)))< 0.01

def set_sess(Sess):
    global S
    S=Sess

def jitter(X):
       X_new=X
       while(isclose(X_new)):
        global seed
        seed=seed+1
        np.random.seed(seed)
```

```
            X_new=X_new+0.2*np.random.randn(get_dim(X,0),1)
        return X_new

def squared_diff(X1,X2):

    l1=get_dim(X1,0); l2=get_dim(X2,0)
    X2_T=tf.transpose(X2)
    X1_mat=tf.tile(X1,[1,l2])
    X2_mat=tf.tile(X2_T,[l1,1])
    K_1=tf.squared_difference(X1_mat,X2_mat)
    return K_1

def abs_diff(X1,X2):

    l1=get_dim(X1,0); l2=get_dim(X2,0)
    X2_T=tf.transpose(X2)
    X1_mat=tf.tile(X1,[1,l2])
    X2_mat=tf.tile(X2_T,[l1,1])
    K_1=tf.abs(tf.sub(X1_mat,X2_mat))
    return K_1

def chol_det(X):
    conditioned=condition(X)
    return tf.square(tf.assert_greater(tf.reduce_prod(tf.diag_part(tf.cholesky(conditioned)))))

def condition(X):
    return (X+tf.transpose(X))/2+tol*np.eye(get_dim(X,0))

def log_det(Z):
    #conditioned=condition(Z)
    Z=(Z+tf.transpose(Z))/2
    return 2*tf.reduce_sum(tf.log(tf.diag_part(tf.cholesky(Z))))

    chol=tf.cholesky(Z)
    logdet=2*tf.reduce_sum(tf.log(tf.diag_part(chol)))
    return logdet

def lambda_Eye(lamb,N):
    Eye=tf.constant(np.eye(N,N), shape=[N,N],dtype=tf.float32)
    sigEye=tf.mul(lamb,Eye)
    return sigEye

### more related to model
def tf_SE_K(X1,X2,len_sc,noise):
    X11=X1
    X22=X2
    sq_diff= squared_diff(X11,X22)
    K=tf.square(noise)*tf.exp(-(0.5*sq_diff/tf.square(len_sc)))#len_sc
    if get_dim(X1,0)==get_dim(X2,0):
        return K+tol*np.eye(get_dim(X1,0))
    else:
        return K

##unit tests so far

#logdet more relevant
def log_det_lemma(ldz,Z_I,U,W_I,V_T):
    LDZ=ldz-log_det(W_I)
    return LDZ+log_det(W_I+Mul(V_T,Z_I,U))

def log_density(x,mu,prec,logdetcov):
    diff=x-mu
    diff_sq=-0.5*tf.matmul(tf.matmul(tf.transpose(diff),prec),diff)
    return diff_sq-0.5*logdetcov
```

```

#corresponds to formula for (z+UWV^T)^-1
def Matrix_Inversion_Lemma(Z_I, U,W_I,V_T):
    A=W_I+Mul(V_T,Z_I,U)
    prod=safe_chol(W_I+Mul(V_T,Z_I,U),Mul(V_T,Z_I))
    return Z_I -Mul(Z_I,U,prod)


def F_bound(y,Kmm,Knm,Knn_trace,sigma):
    #matrices to be used
    N=get_dim(Knn,0)
    sig_sq=tf.square(sigma)
    sigEye=lambda_Eye(sig_sq,N)
    sigEye_I=lambda_Eye(1/sig_sq,N)
    zeros=tf.constant(np.zeros(N),shape=[N,1],dtype=tf.float32)
    Kmn=tf.transpose(Knm)
    #main calcz
    prec=Matrix_Inversion_Lemma(sigEye_I,Knm,Kmm,Kmn)
    log_det_cov=log_det_lemma(tf.log(sig_sq)*N,sigEye_I,Knm,Kmm,Kmn)
    log_den=log_density(y,zeros,prec,log_det_cov)
    trace_term=Knn_trace-tf.trace(Mul(Knm,safe_chol(Kmm,Kmn)))
    return log_den-trace_term
```

B.2 Natural Gradients Implementation

B.3 First DGP Implementation

```
import maxKern as maxKern
import maxKL as KL
import maxKernStats as kstat
import maxHelp as Help
import maxTypes as ty
import tf_hacks as tfh
import kullback_leiblers as kl2

tol =1e-2


import tensorflow as tf
import numpy as np


class basicLayer(object):
    def __init__(self, l, ind, outd, n, m):
        self.l=l
        self.ind=ind; self.outd=outd
        self.n=n; self.m=m

        self.f_sigma=tf.Variable(tf.ones([1],dtype=tf.float64),dtype=tf.float64)
        self.f_beta=tf.square(self.f_sigma)
        self.ps_P=ty.points(num=m,dim=outd)
        self.prev=None; self.post=None

        self.mu=ty.mu(num=m,dim=outd)
        self.sigma=ty.full_cov(ind=outd,squareDim=m)

        kern = dict()
        for i in xrange(0, outd):
            kern[i] = maxKern.RBF(ind)
        self.kern=kern

    def Kuu(self,i):
        A=self.kern[i].K(self.ps_P.getNxD()) +tol*tfh.eye(self.m)
        return(A+tf.transpose(A))/2


    def Knn(self,i,data,data2=None):
        assert Help.get_dim(data,1) == self.ind
        n=Help.get_dim(data,0)
        if data2==None:
            A=self.kern[i].K(data) +tol*tfh.eye(n)
            return (A+tf.transpose(A))/2
        else:
            assert Help.get_dim(data2,1) == self.ind
            return self.kern[i].K(data,data2)

    def KL(self):
        KL_div=0
        for d in xrange(0,self.outd):
            mu_d= self.mu.getDim(d,[-1,1])
            KL_div+=kl2.gauss_kl(-mu_d,tf.expand_dims(tf.cholesky(self.sigma.get_dim(d)),-1),
              self.Kuu(d),1)
        return KL_div

    def post_sample(self,data):
        assert Help.get_dim(data,0) == self.n
        assert Help.get_dim(data,1) == self.ind
        output = []


    # Should we sample one point at a time? ...return
    def sample(self, data):
        n=Help.get_dim(data,0)
        assert Help.get_dim(data,1) == self.ind
        final=[]
        output = []
        for d in xrange(self.outd):
            K_uu_chol=tf.cholesky(self.Kuu(d))
            K_nu=self.Knn(d, data,self.ps_P.getNxD())
            K_un=tf.transpose(K_nu)
            mean=tf.matmul(K_nu,tf.cholesky_solve(K_uu_chol,self.mu.getDim(d)))
            factor=tf.cholesky_solve(K_uu_chol,K_un)
            var=tfh.eye(n)*tf.inv(self.f_beta)+self.kern[d].Kdiag(data)\
                -tf.diag(tf.diag_part(Help.Mul(K_nu,factor)))+\
            tf.diag(tf.diag_part(Help.Mul(tf.transpose(factor),self.sigma.get_dim(d),factor)))
            var_chol=tf.sqrt(var)
            noise=tf.random_normal(shape=[self.n,1],mean=0.0,stddev=1.0,
                                    dtype=tf.float64)
            output.append(tf.squeeze(tf.matmul(var_chol,noise)+mean))
        final=tf.transpose(tf.pack(output))

        return final #+ post_noise


class oneLayer(basicLayer):
    def __init__(self, l, ind, outd, n, m,input, output):
        basicLayer.__init__(self,l,ind,outd,n,m)
        self.dataPoints = input
        self.M=ty.mean(num=n, dim=outd,data=output)
        self.S=ty.diag_cov(num=n, dim=outd,data=tf.zeros([n,outd]))

    def psi(self,d):
        Z=self.ps_P.getNxD();  kern=self.kern[d];M=self.dataPoints
        var=kern.variance
        psi0 = tf.cast(self.n, tf.float64) * var
        psi1=self.Knn(d,M,Z)
        psi2=tf.matmul(tf.transpose(psi1),psi1)
        return psi0, psi1,psi2

    def L_term(self):
        # change minus
        result=-0.5*self.f_beta*(tf.reduce_sum(tf.square(self.M.getNxD())))
        # 1st half of Line 1
        result-=0.5*tf.log(2*np.pi*tf.inv(self.f_beta))*self.n*self.outd
        #Line 2 and 3
        for j in xrange(self.outd):
            # Definitions
            zeta_l, psi_l, phi_l=self.psi(j)
            assert(Help.get_dim(psi_l,0)==self.n)
            assert(Help.get_dim(psi_l,1)==self.m)

            mu_j=self.mu.getDim(j,[-1,1])
            M_j=self.M.getDim(j,[1,-1])
            Kuu_chol=tf.cholesky(self.Kuu(j));

            #term (3).. changed minus sign..check
            result+=self.f_beta*tf.reduce_sum(tf.squeeze(M_j)*tf.squeeze(tf.matmul(psi_l,tf.
              cholesky_solve(Kuu_chol,mu_j))))
            #term (4)..ok
            result-=0.5*self.f_beta*(zeta_l-tf.trace(tf.cholesky_solve(Kuu_chol,phi_l)))
            #term (5)
            cov=tf.matmul(mu_j,tf.transpose(mu_j))+self.sigma.get_dim(j)
            result-=0.5*self.f_beta*tf.trace(Help.Mul(tf.cholesky_solve(Kuu_chol,cov),tf.
              cholesky_solve(Kuu_chol,phi_l)))

        return result

    def bound(self):
        return self.L_term()
```

B.4 Natural Gradient Implementation (as part of a DGP implementation)

```
import tensorflow as tf
import densities as dens
import tf_hacks as tfh
import maxKern as kern
import maxKernStats as kstat
import maxHelp as Help
import maxTypes as mT
import numpy as np

tol = 1e-4

class basicLayer(object):

    def __init__(self,n,m):
        self.n=n; self.m=m
        self.hS = 0.0001*tf.constant(np.square(np.random.randn(n, 1)), dtype=tf.float64)
        self.hMu = tf.Variable(tf.random_uniform([n, 1],minval=-1,maxval=1, dtype=tf.float64))
        self.variance = 4*tf.exp(tf.Variable(tf.ones([1],dtype=tf.float64)))
        self.lengthscale = tf.ones([1],dtype=tf.float64)
        self.Kern = kern.Kern(1, self.variance, self.lengthscale)
        self.beta=0.01*tf.square(tf.ones([1],dtype=tf.float64))
        self.pseudo = tf.Variable(tf.reshape(tf.linspace(tf.constant(-1.0, dtype=tf.float64),
          1.0, num = m),[-1,1]))
        self.psS = 0.01*tfh.eye(m)
        self.psMu = 2*tf.constant(np.random.randn(m,1), dtype=tf.float64)
        #self.varList=[self.sigma,self.std,self.lengthscale,self.pseudo]
        self.Kmm = self.Kern.K(self.pseudo, self.pseudo)
        self.Kmm_chol = tf.cholesky(self.Kmm)
        self.KmmInvM =tf.cholesky_solve(self.Kmm_chol, self.psMu)

    def lBound(self, TrKnn, Knm, Kmnnm):
        Kmm = self.Kern.K(self.pseudo, self.pseudo)
        Kmm_chol = tf.cholesky(Kmm)
        KmmInvM =tf.cholesky_solve(Kmm_chol, self.psMu)

        betaM = tfh.eye(self.n)/self.beta
        t11 = tf.reduce_sum(tf.square(self.hMu) + self.hS)
        t12 = Help.Mul(tf.transpose(self.hMu), Knm, KmmInvM)
        t13 = tf.trace(tf.matmul(tf.cholesky_solve(Kmm_chol, Kmnnm), tf.cholesky_solve(Kmm_chol
          , tf.matmul(self.psMu, tf.transpose(self.psMu)) + self.psS))
        term1 = - 0.5 * self.beta * t11 + self.beta * t12 - 0.5 * self.beta * t13
        zeros = tf.zeros([self.n,1], dtype=tf.float64)

        L3 = tf.log(dens.multivariate_normal(zeros, zeros, tf.cholesky(betaM)))
        L3 = L3 + term1
        L3 = L3 - 0.5 * self.beta * (TrKnn - tf.trace(tf.cholesky_solve(Kmm_chol, Kmnnm)))
        L3 = L3 - 0.5 * tf.trace(self.beta * tf.matmul(tf.cholesky_solve(Kmm_chol, Kmnnm), tf.
          cholesky_solve(Kmm_chol, tfh.eye(self.m))))
        L3 = L3 - self.gauKL(self.psMu, self.psS, tf.zeros([self.m, 1], dtype=tf.float64), Kmm)
        return L3

    def gauKL(self,pm, pv, qm, qv):
        # KL(P, Q) --> P on bottom
        delta = qm - pm
        qv_chol = tf.cholesky(qv)
        gkl = tf.log(tf.matrix_determinant(qv))
        gkl = gkl - tf.log(tf.matrix_determinant(pv))
        gkl = gkl + tf.trace(tf.cholesky_solve(qv_chol, pv))
        gkl = gkl + tf.matmul(tf.transpose(delta), tf.cholesky_solve(qv_chol, delta))-self.m
        return 0.5 * gkl

    def lUpdateMuS(self,step,Knm,Knmmn):
        tolMat=tfh.eye(self.m)*tol
        Kmn = tf.transpose(Knm)
        Kmm_chol = tf.cholesky(self.Kern.K(self.pseudo, self.pseudo))
        KmmInvKmn = tf.cholesky_solve(Kmm_chol+tolMat, Kmn)
        KmmInv = tf.cholesky_solve(Kmm_chol+tolMat, tfh.eye(self.m))
        S_chol = tf.cholesky(self.psS)

        lamb = self.beta * Help.Mul(KmmInv, self.Kmnnm, KmmInv) + KmmInv
        SInv = tf.cholesky_solve(S_chol+tolMat, tfh.eye(self.m))

        theta2 = -0.5 * SInv + step * (-0.5 * lamb + 0.5 * SInv)
        SInvMu = tf.cholesky_solve(S_chol+tolMat, self.psMu)
        theta1 = SInvMu + step * (tf.matmul(self.beta * KmmInvKmn, tf.reshape(self.hMu, [self.n
          ,1])) - SInvMu)
        return self.getMuS(theta1, theta2)

    def getMuS(self,th1, th2):
        tolMat=tfh.eye(self.m)*tol
        outMu = tf.reshape(-0.5 * tf.matrix_solve(th2+tolMat, th1), [self.m,1])
        outS = -0.5 * tf.matrix_solve(th2+tolMat, tfh.eye(self.m))
        return outMu, outS

    def pred(self,X):
        Kxm=self.Kern.K(X,self.pseudo)
        Kmm=self.Kern.K(self.pseudo,self.pseudo)
        return  tf.matmul(Kxm,tf.matrix_solve(Kmm+tol*tfh.eye(self.m),self.psMu))

    def pipedPred(self):
        return self.pred(self.X)

    def safe_solve_chol(self,A,B):
        safety=(A+tf.transpose(A)+tol*tfh.eye(self.m))/2
        chol_safety=tf.cholesky(safety)
        return tf.cholesky_solve(chol_safety,B)
    def safe_solve(self,A,B):
        return tf.matrix_solve(A+tol*tfh.eye(self.m),B)

################################################################################

class firstLayer(basicLayer):

    def __init__(self, n, m, X,Y):
        basicLayer.__init__(self, n, m)
        self.hMu=3*tf.Variable(Y)/4
        self.hS=0.001*tf.constant(np.square(np.random.randn(self.n, 1)), dtype=tf.float64)
        self.X=X
        self.Knm = self.Kern.K(self.X, self.pseudo)
        self.Kmnnm = tf.matmul(tf.transpose(self.Knm), self.Knm)
        self.TrKnn = tf.trace(self.Kern.K(self.X, self.X))

    def bound(self):
        return self.lBound(self.TrKnn, self.Knm, self.Kmnnm)

    def updateMuS(self, step):
        self.psMu, self.psS = self.lUpdateMuS(step, self.Knm, self.Kmnnm)


################################################################################

class midLayer(basicLayer):

    def __init__(self, layer, n, m, data):
        basicLayer.__init__(self, layer, n, m)
        self.TrKnn, self.Knm, self.Kmnnm = self.psi()


    def bound(self):
        return self.lBound(self.TrKnn, self.Knm, self.Kmnnm)

    def updateMuS(self, step):
        self.psMu, self.psS = self.lUpdateMuS(step, self.Knm, self.Kmnnm)

    def psi(self):
        return kstat.build_psi_stats_rbf(self.pseudo, self.Kern, self.prev.hMu, self.prev.hS)

    def setTrKnnKnmKmnnm(self):
        self.TrKnn, self.Knm, self.Kmnnm = self.psi()


################################################################################

class lastLayer(basicLayer):

    def __init__(self, n, m, X,Xvar,Y):
        basicLayer.__init__(self, n, m)
```

```python
        self.hMu = Y/2
        self.hS = tf.zeros([self.n,1],dtype=tf.float64)
        self.Xvar=Xvar
        self.X=X
        self.TrKnn, self.Knm, self.Kmnnm = self.psi()

    def bound(self):
        #self.TrKnn, self.Knm, self.Kmnnm = self.psi()
        return self.lBound(self.TrKnn, self.Knm, self.Kmnnm)

    def updateMuS(self, step):
        #self.TrKnn, self.Knm, self.Kmnnm = self.psi()
        self.psMu, self.psS = self.lUpdateMuS(step, self.Knm, self.Kmnnm)

    def psi(self):
        return kstat.build_psi_stats_rbf(self.pseudo, self.Kern, self.X, self.Xvar)

    def reset(self,value,session):
        job=self.X.assign(value)
        session.run(job)

    def predict_new(self):
        #self.TrKnn, self.Knm, self.Kmnnm = self.psi()
        Kmm=self.Kern.K(self.pseudo,self.pseudo)
        return  tf.matmul(self.Knm,tf.matrix_solve(Kmm+tol*tfh.eye(self.m),self.psMu))







#################################################################################



class hybrid(basicLayer):

    def __init__(self, layer, n, m, X, Y):
        basicLayer.__init__(self, layer, n, m)
        self.X = X
        self.Y = Y
        self.hMu = Y
        self.hS = tf.zeros([self.n, 1], dtype=tf.float64)
        self.Knm = self.Kern.K(self.X, self.pseudo)
        self.Kmnnm = tf.matmul(tf.transpose(self.Knm), self.Knm)
        # TODO: Don't compute Knn
        self.TrKnn = tf.trace(self.Kern.K(self.X, self.X))

    def bound(self):
        return self.lBound(self.TrKnn, self.Knm, self.Kmnnm)



    def updateMuS(self, step):
        self.psMu, self.psS = self.lUpdateMuS(step, self.Knm, self.Kmnnm)
```

B.5 Second DGP Implementation using GPflow

```python
#DGP Implementation borrowing routines from GPflow
```

```
import tensorflow as tf
import numpy as np
from .model import GPModel
from .param import Param, DataHolder
from .mean_functions import Zero
from . import likelihoods
from .tf_hacks import eye
from . import transforms
from . import kernels as ke
from . import kernel_expectations as ks

class SGPR(GPModel):

    def __init__(self, X, Y, mean_function=Zero()):
        self.D = X.shape[1]
        self.N = X.shape[0]
        self.M=20

        X1 = DataHolder(X, on_shape_change='pass')
        Y1 = DataHolder(Y, on_shape_change='pass')
        self.X=X1;self.Y=Y1;
        self.kern1 = ke.RBF(self.D)
        self.likelihood1 = likelihoods.Gaussian()
        self.likelihood2= likelihoods.Gaussian()
        GPModel.__init__(self, X, Y, self.kern1, self.likelihood1, mean_function)
        self.kern2= ke.RBF(self.D)
        ND=np.ones((self.N,self.D))
        MD = np.reshape(np.linspace(0, 1, (self.M*self.D)),(self.M,self.D))
        self.X2 = Param(Y/2)
        self.X2_var = Param(ND, transforms.positive)
        self.Z2 = Param(MD)
        self.Z = Param(MD)

    def build_likelihood_gplvm(self):
            """
            Construct a tensorflow function to compute the bound on the marginal
            likelihood.
            """
            Y = self.Y;
            Z = self.Z2;
            K = self.kern2;
            X = self.X2
            X_var=self.X2_var;

            num_inducing = tf.shape(Z)[0]

            psi0, psi1, psi2 = ks.build_psi_stats(Z, K, X, X_var)
            self.psi0=psi0
            self.psi1=psi1
            self.psi2=psi2
            Kuu = K.K(Z) + eye(num_inducing) * 1e-6
            L = tf.cholesky(Kuu)
            sigma2 = self.likelihood2.variance
            sigma = tf.sqrt(sigma2)

            # Compute intermediate matrices
            A = tf.matrix_triangular_solve(L, tf.transpose(psi1), lower=True) / sigma
            tmp = tf.matrix_triangular_solve(L, psi2, lower=True)
            AAT = tf.matrix_triangular_solve(L, tf.transpose(tmp), lower=True) / sigma2
            B = AAT + eye(num_inducing)
            LB = tf.cholesky(B)
            log_det_B = 2. * tf.reduce_sum(tf.log(tf.diag_part(LB)))
            c = tf.matrix_triangular_solve(LB, tf.matmul(A, Y), lower=True) / sigma

            # KL[q(x) || p(x)]
            NQ = tf.cast(tf.size(X), tf.float64)
            D = tf.cast(tf.shape(Y)[1], tf.float64)
            KL = -0.5 * tf.reduce_sum(tf.log(X_var))

            # compute log marginal bound
            ND = tf.cast(tf.size(Y), tf.float64)
            bound = -0.5 * ND * tf.log(2 * np.pi * sigma2)
            bound += -0.5 * D * log_det_B
            bound += -0.5 * tf.reduce_sum(tf.square(Y)) / sigma2
            bound += 0.5 * tf.reduce_sum(tf.square(c))
            bound += -0.5 * D * (tf.reduce_sum(psi0) / sigma2 -
                                 tf.reduce_sum(tf.diag_part(AAT)))
            bound -= KL

            return bound

    def build_likelihood_SGPR(self):
        """
        Construct a tensorflow function to compute the bound on the marginal
        likelihood. For a derivation of the terms in here, see the associated
        SGPR notebook.
        """
        Y_var=self.X2_var
        Y=self.X2; Z=self.Z; K=self.kern1;  X=self.X
        num_inducing = tf.shape(Z)[0]
        num_data = tf.cast(tf.shape(Y)[0], tf.float64)
        output_dim = tf.cast(tf.shape(Y)[1], tf.float64)

        Kdiag =K.Kdiag(X)
        Kuf = K.K(Z, X)
        Kuu = K.K(Z) + eye(num_inducing) * 1e-6
        L = tf.cholesky(Kuu)
        sigma = tf.sqrt(self.likelihood1.variance)

        # Compute intermediate matrices
        A = tf.matrix_triangular_solve(L, Kuf, lower=True) / sigma
        AAT = tf.matmul(A, tf.transpose(A))
        B = AAT + eye(num_inducing)
        LB = tf.cholesky(B)
        Aerr = tf.matmul(A, Y)
        c = tf.matrix_triangular_solve(LB, Aerr, lower=True) / sigma

        # compute log marginal bound
        bound = -0.5 * num_data * output_dim * np.log(2 * np.pi)
        bound += - output_dim * tf.reduce_sum(tf.log(tf.diag_part(LB)))
        bound -= 0.5 * num_data * output_dim * tf.log(self.likelihood1.variance)
        bound += -0.5*tf.reduce_sum(tf.square(Y))/self.likelihood1.variance
        bound += 0.5*tf.reduce_sum(tf.square(c))
        bound += -0.5 * tf.reduce_sum(Kdiag) / self.likelihood1.variance
        bound += 0.5 * tf.reduce_sum(tf.diag_part(AAT))
        bound -= 0.5*tf.reduce_sum(Y_var)/self.likelihood1.variance

        return bound

    def build_likelihood(self):
        bound= self.build_likelihood_SGPR()
        bound+=self.build_likelihood_gplvm()
        return bound

    def pred(self, X, u, mu,K):
        Kmm = K.K(u, u)+eye(self.M) * 1e-6
        Knm = K.K(X, u)
        posterior_mean = tf.matmul(Knm, tf.matrix_solve(Kmm, mu))
        Knn = K.K(X, X)
        full_cov = Knn - tf.matmul(Knm, tf.matrix_solve(Kmm, tf.transpose(Knm)))
        return posterior_mean, full_cov



    def build_predict(self):
        Y = self.Y;
        Z = self.Z2;
        K = self.kern2;
        X = self.X2
        X_var = self.X2_var;
        Kmm=K.K(Z,Z)+ eye(self.M) * 1e-6
        Kmn = K.K(Z, X)
        psi0, psi1, Kmnnm = ks.build_psi_stats(Z, K, X, X_var)
        sigma2=self.likelihood2.variance
        A_I = Kmnnm/sigma2 + Kmm
        Sig = tf.matmul(Kmm, tf.matrix_solve(A_I, Kmm))
        mu =  tf.matmul(Kmm, tf.matrix_solve(A_I, tf.matmul(Kmn, Y)))/sigma2
        return mu, Sig

    def build_other(self):
        mu, Sig=self.build_predict()
        mean, cov=self.pred(self.X2,self.Z2,mu,self.kern2)
        return mean, tf.diag_part(cov)
```