Designing Neural Network Hardware Accelerators Using Deep Gaussian Processes

Márton Havasi

Supervisor: Dr. J. M. Hernández-Lobato

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy

Churchill College August, 2017

Declaration

I, Marton Havasi of Churchill 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: 14007

Márton Havasi
August, 2017

Acknowledgements

First and foremost, I would like to express my gratitude towards my supervisor, Dr. J. M. Hernández-Lobato, for his help and support. His immense knowledge, guidance and insightful comments proved incredibly valuable throughout the project. It has been a pleasure working with him.

Secondly, I want to thank all the MLSALT professors, lecturers and examiners who have been there to answer my questions and ready to provide insights over the course of the year. They have done a great job of teaching and explaining the intricacies of machine learning and speech technology.

Finally, I would like to thank Microsoft Research for sponsoring this project. The Azure computing resources proved extremely useful when I was running my costly experiments.

Abstract

Neural networks have become the mainstream tool for classification tasks in the past decade. However, their spread to consumer devices, such as IoT devices, smartphones and laptops, has been hindered by their high requirements for processing power. A solution to the problem is to use hardware accelerators specifically designed for deep neural networks in order to increase their efficiency.

This project aimed to tackle the problem of optimizing aforementioned designs. The goal was to find configurations that struck a balance between power consumption and error rate. Since testing a configuration can be expensive, we treated the design process as a black-box multiobjective optimization problem with the objectives being low prediction error rate and low power consumption.

The underlying predictive model for the optimization process was a Deep Gaussian Process (DGP). DGPs were ideal candidates for the task at hand, because they give reasonable uncertainty estimates even in the presence of limited training data. In terms of flexibility, a DGP is equivalent to an infinitely wide, multi-layer neural network.

Moreover, our novel extension to DGPs were Joint DGP models. In Joint DGP models, the same network is used to model both objective functions. There are two advantages to this setup. Firstly, the number of models that need to be trained is reduced to one. Secondly, this configuration allows the deep model to capture the underlying correlation between the objective functions which can potentially lead to an increase in performance.

We found that DGPs performed similarly to the baseline Gaussian Process (GP) model, both of which significantly outperformed the naïve data collection strategy. Furthermore, we found evidence of improved uncertainty estimates in the case of DGP and Joint DGP models.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Designing neural network hardware accelerators is a slow and expensive process. Each accelerator design takes multiple iterations of testing and adjustments to perfect. In particular, the testing of hardware configurations is resource and time consuming since the most reliable way to get accurate metrics is to emulate the hardware under realistic loads. In this project, we seek to reduce the number of testing iterations necessary in the design of a high-performance hardware accelerator.

Each accelerator design involves many parameters such as memory size, cache bandwidth, floating-point precision and clock-frequency. The space spanned by these parameters is simply too large to trust a human expert with setting all the parameters. This project tackles the final stage of hardware design: tuning these parameters in order to optimize the performance of the hardware accelerator.

When optimizing the design, there are two conflicting goals in mind. Both the error rate of the neural network and the power consumption of the hardware nees to be low. Furthermore, due to the costs of testing, the number of configurations we can evaluate is limited.

We approached the process of designing neural network hardware accelerators as a black-box multiobjective optimization problem. Using a limited budget of evaluations, we wish to approximate the set of 'optimal' configurations (also called Pareto points), that is, configurations that are not worse in both error rate and power consumption than any other configuration. The process went as follows. In each iteration, we tested a new configuration. Then, using the result of the test, we updated our model's belief of the behavior of all the candidate configurations. At the end of the iteration, the model produced an optimized decision about the location of our next evaluation. We had two goal with each evaluation. Firstly, to find optimal configurations, secondly, to improve the accuracy of our beliefs thus solving the exploration vs. exploitation problem.

The efficiency of the process was judged by how well it managed to approximate the optimal set of points. This was achieved by maximizing the metric, which was defined as the hypervolume enclosed by the Pareto points and a fixed reference point as shown in Figure 1.1.

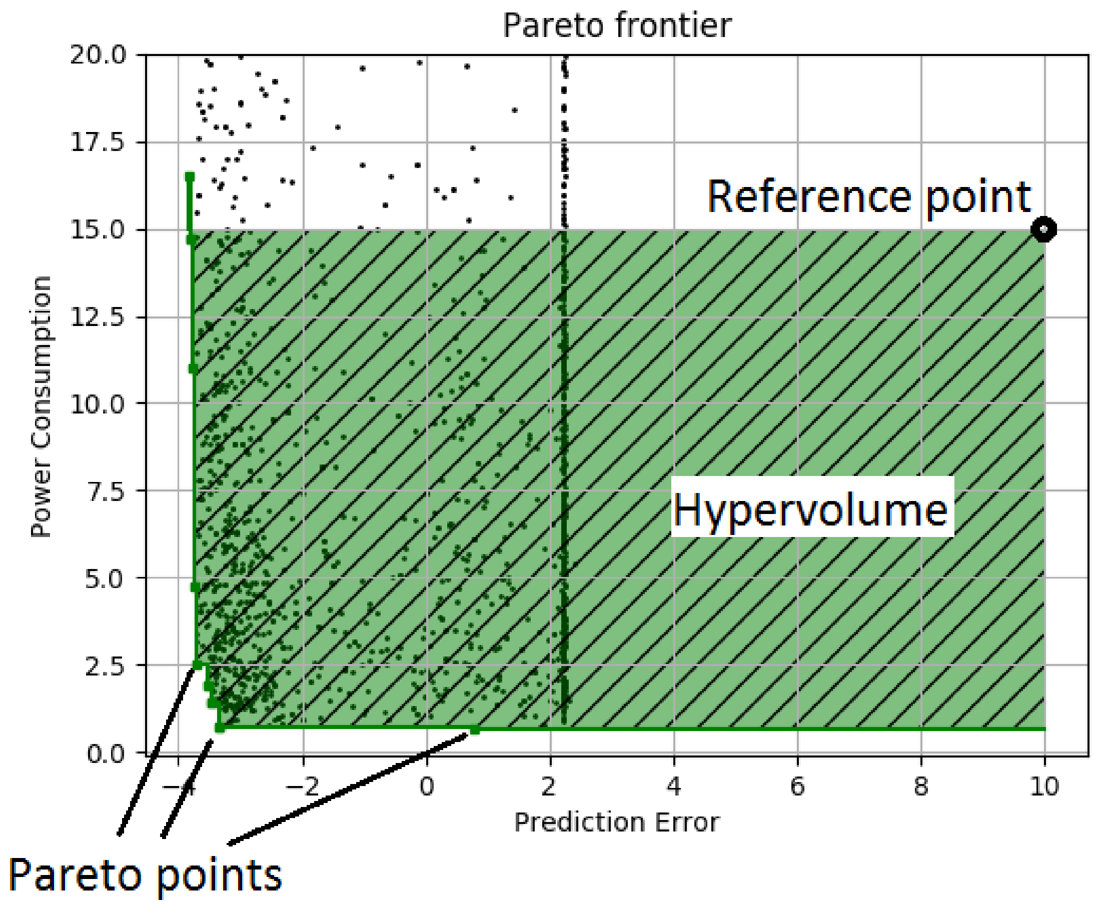


Fig. 1.1 The hypervolume enclosed by the Pareto points and the reference point. The black markers denote all the candidate configurations.

As the underlying model for the optimization process, we used Deep Gaussian Processes (DGP) (Damianou and Lawrence, 2013) with Gaussian Processes (GP) (Williams and Rasmussen, 1996) providing the baseline. They were ideal candidates for the task at hand, since they are flexible and they can provide sensible uncertainty estimates even from a few datapoints.

The implementation and training of the DGP models was based on (Bui et al., 2016). We employed sparsity approximations in order to improve scalability and used Expectation Propagation for computationally tractable training.

Furthermore, we investigated a novel extension to DGPs. In the case of DGPs, a separate model was trained to predict each objective function. However, in Joint DGPs, we used the same network to predict all objective functions at the same time. In this setup, the hidden layers were shared between the objective functions, but each objective had its own output node for prediction. The rationale behind this setup was that it allowed the model to capture the underlying correlations between the objective functions in the hidden layers. This lead to slightly improved uncertainty estimates over DGPs as well as it reduced the number of models that needed to be trained to one.

We derived two conclusions from our experiments. First, DGPs performed on par with GPs over the course of the optimization process and they both considerably outperformed the naïve (random) data-collection strategy. Second, the added flexibility of DGPs and Joint DGPs increased the quality of the uncertainty estimates. The test log-likelihood of the GP model was -1.20 ± 0.06 as opposed to -0.61 ± 0.04 of DGPs and -0.48 ± 0.05 of Joint DGPs at 300 training points.

1.1 Work completed

In the first part of the project, we implemented our DGP model based on (Bui et al., 2016) in Tensorflow. This model used sparse GP approximations for scalability and Expectation Propagation for training. The technical details are described in Section 3.2 and the implementation details in Section 3.3.1.

Following that, we built a framework for multiobjective optimization. Our choice of data-collection strategy was SMSego (Section 3.1) which we tested in combination with the different predictive models.

The data used in the experiments consisted of 6276 randomly sampled points sourced by simulations of neural network hardware accelerators (Reagen et al., 2016).

Finally, we conducted experiments regarding the improvement in hypervolume over the course of the optimization process, as well as separate experiments for testing the predictive power of each model. Our findings along with the discussion of the results are included in Chapter 5.

Chapter 2

Literature review

This chapter introduces Bayesian Optimization (BO) and Multiobjective Optimization including a summary of the current state-of-the-art data collection strategies. This is followed by a description of Gaussian Processes (GP) and a review of Deep Gaussian Processes (DGP). The chapter is concluded with an overview of recent works related to the application of Bayesian Optimization to hardware design problems.

2.1 Bayesian Optimization

With the increasing complexity of product designs, such as websites, microprocessors or machine learning models, a new area of problems emerged. These designs often had a vast number or parameters that could interact in an unexpected and unpredictable ways. Far too many for a human expert to decide upon. Bayesian Optimization (BO) (Shahriari et al., 2016) emerged as a solution to the problem. Bayesian Optimization is a technique that uses machine learning models in order to tune the parameters of a configuration which often results in significant improvement in efficiency.

In technical terms, Bayesian Optimization refers to a family of approaches to sequential optimization of an objective function where the analytic form of the function is unknown (ie. black-box function) and the cost of evaluating the objective function is expensive. Typically, the number of evaluations is restricted due to the high cost.

$$x' = \operatorname*{argmax}_{x \in \mathcal{X}} f(x)$$

The key idea is that one can invest significant computational effort into intelligent decision making under the assumption that the cost of evaluating the expensive objective function dominates the overall cost.

2.1.1 Single objective optimization

Single objective optimization is the most common case where there is only a single objective function to maximize. Examples include maximizing the number of visitors to a website, maximizing consumer satisfaction, minimizing the power consumption of a circuit and maximizing the test log-likelihood of a statistical model.

The process requires two pieces. A predictive model and a data collection strategy.

The predictive model must be able to estimate the objective function with uncertainty bounds. It maintains a probabilistic belief about the behavior of the objective function as the optimization progresses. Due to the low number of samples, it is key to have a model that gives reasonable estimates even from only a few datapoints. The most commonly used model, a Gaussian Process, is described in Section 2.2 followed by a description of Deep Gaussian Processes in Section 2.3.

The data collection strategy takes the form of an acquisition function that approximates how much 'value' one can gain by evaluating the objective function at a certain point. In each iteration, the acquisition function is calculated over all the points and the highest scoring point is selected for evaluation.

The iterative process is shown on Algorithm 1.

```
$f \leftarrow$ objective function
$\mathcal{M} \leftarrow$ predictive model
$\alpha(x, \mathcal{M}) \leftarrow$ acquisition function
Data: $\mathcal{X} \leftarrow$ is the set of candidate parameter configurations (grid)
Data: $\mathcal{D}_0 = \{\}$
for $n = 1, 2 \ldots, N$ do
    $x_n = \text{argmax}_{x \in \mathcal{X}} \alpha(x, \mathcal{M}(\mathcal{D}_{n-1}))$
    $y_n = f(x_n)$
    $\mathcal{D}_n = \{(x_n, y_n)\} \bigcup \mathcal{D}_{n-1}$
end
Result: $x_{\text{argmax}_n(y_n)}$
```

Algorithm 1: Bayesian Optimization

The acquisition function is responsible for balancing exploration (improving the accuracy of the predictive model) versus exploitation (evaluating points that are likely to be optimal) using the estimates and uncertainties provided by the predictive model. The exploration part is key for long-term improvement while exploitation reaps the immediate benefits of the predictive model.

The most commonly used acquisition functions (Jones et al., 1998) include:

- **Probability of improvement (PI):**

$$\alpha(x) = P(y \geq max_n(y_n)|x)$$

- **Expected improvement (EI):**

$$\alpha(x) = E\left(max(y - max_n(y_n)|x), 0.0\right)$$

- **Lower Confidence Bound (LCB):**

$$\alpha(x) = \mu_y - c\sigma_y^2$$

  Where (μ_y, σ_y^2) is the model prediction for y and c is a constant.

- **Entropy Search (ES):**

$$\alpha(x) = H(p(x'|\mathcal{D})) - H(p(x'|\mathcal{D} \bigcup \{(x,y)\}))$$

  Where H denotes the entropy and x' denotes the optimal solution. The acquisition value is the reduction in the entropy of the solution.

Each of these acquisition functions achieve the goal of balancing exploration versus exploitation in some way. Their degree of effectiveness varies, although we can single out Entropy Search as being the most efficient for a wide variety of problems.

2.1.2 Constrained optimization

While Bayesian Optimizations is very successful at optimizing a single objective, in the real world, it is rarely the goal to optimize a single objective at all costs. Typically, the parameters are constrained in some way. For instance, one might want to maximize the number of visitors on the website, but there is an upper bound to the server costs the he or she is willing to pay. A second example is the design of chemical molecules. The constraints arise from invalid combination of parameters that lead to molecules that cannot be synthesized (Gómez-Bombarelli et al., 2016).

Formally in constrained optimization, we aim to maximize $f(x)$ subject to constraints $c_1(x), \ldots, c_K(x)$.

$$x' = \underset{x \in \mathcal{X}}{\operatorname{argmax}} f(x) \quad \text{s.t.} \quad c_1(x) \geq 0, \ldots, c_K(x) \geq 0$$

The approach in this case is quite similar to Bayesian Optimization (Algorithm 1), although the acquisition functions need to be adapted to the constrained case.

Some methods, such as PI and EI (Jones et al., 1998) can be straight-forwardly applied in the constrained case. Simply, train a probabilistic model for each of the constraints $c_1(x), \ldots, c_K(x)$ and incorporate the probability of an input x meeting all of them in the calculation of the acquisition value.

For example, PI becomes:

$$\alpha(x) = P\left(c_1(x) \geq 0, \ldots, c_K(x) \geq 0, y \geq max_n(y_n) | x\right)$$

However, for both PI and EI, it can be an issue if there has not been a valid solution y_n in the process so far. Predictive Entropy Search with Constraints (PESC) (Hernández-Lobato et al., 2015) deals with the issue using an information-based approach which was shown to outperform both EI and PI-based methods on real-world examples.

In addition, in an inspiring paper (Hernández-Lobato et al., 2016b) PESC was extended to support decoupled constraint evaluation. This can be extremely useful in cases that allow separate evaluation of the constraints. A good example is that in molecule design, it is often much quicker to decide whether a design leads to a synthesizable molecule than it is to predict its properties. In these cases, decoupled evaluations can lead to significant cost reduction.

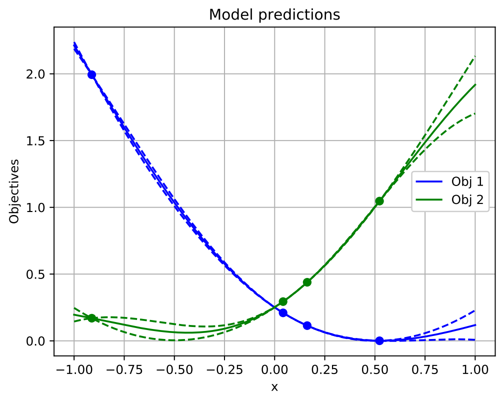
2.1.3 Multiobjective Optimization

A natural extension of constrained optimization is to have multiple objective functions instead of constraints. For instance, one might be interested in all hardware configurations that have high performance while having the lowest possible power consumption for the given performance.

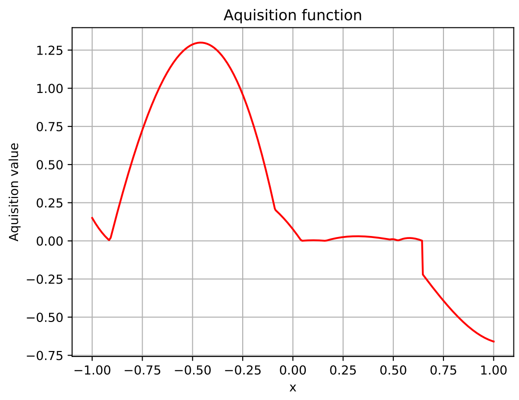
Multiobjective Optimization (or Pareto optimization) is concerned with optimizing multiple, back-box objective functions simultaneously. In non-trivial cases, the objectives are conflicting and therefore there is no single optimal solution to be found. Instead, the solution consists of a set of optimal points.

A point is Pareto-optimal when there is no single point that performs better than said point in every objective function. In the optimization process we seek to approximate the set of Pareto-optimal points. The set of Pareto-optimal points is called the Pareto-front.

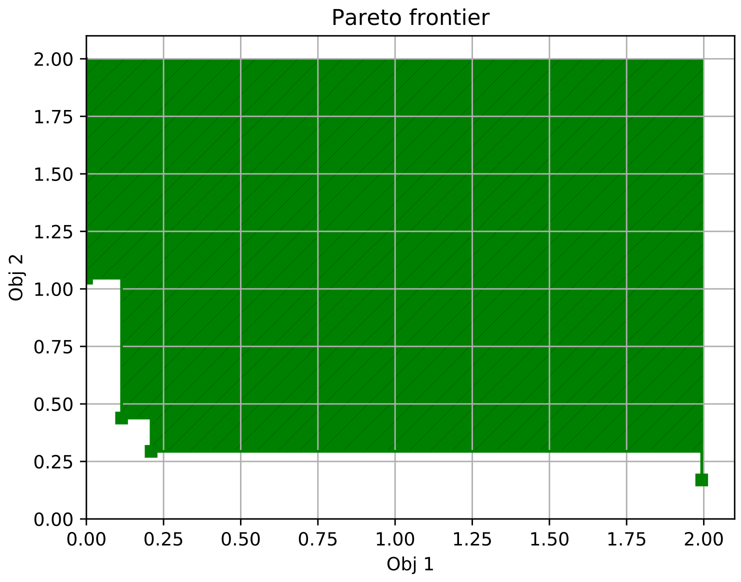
The optimization process is analogous to BO with the exception that a separate predictive model is required for each objective function. Figure 2.1 demonstrates the concept of the predictive model, acquisition function and the hypervolume for a toy problem.



(a) The 5 datapoints and the predictions made by the predictive model.



(b) The acquisition value.



(c) The hypervolume of the current Pareto-front.

Fig. 2.1 Demonstration of the Multiobjective Optimization process on a toy problem.

Hypervolume

When comparing two sets of Pareto-front approximations, it is not necessarily the case that one is strictly better than the other. It might be the case that both sets contain points that dominate points from the other set (domination is when one point outperforms another in every objective). Therefore it is necessary to introduce a performance metric that assesses the quality of the Pareto-front approximations.

The performance metric that is commonly used in multiobjective optimization is the hypervolume of the Pareto-front. The hypervolume is defined as the volume enclosed by the Pareto front and a fixed reference point in the objective space. The intuition is that the process aims to maximize the volume of the dominated points, which is maximal when the optimal Pareto-frontier has been found. Figure 1.1 demonstrates the concept for two objectives. The reference point is typically chosen to be the *maximal value + 1* in each objective function (Emmerich and Naujoks, 2004), although we found that an inappropriate reference point can have detrimental impact on the performance (Section 5.1).

Acquisition functions

Multiobjective optimization is inherently different from single objective Bayesian optimization due to having no single objective function to maximize. The acquisition functions from the single-objective case are not straight-forwardly applicable in the multiobjective case.

In this project, the method we used was SMSego which is described in detail in Section 3.1.

Below is a brief overview of the current state-of-the-art data collection strategies for the multiobjective case. Some of the analysis is based on (Hernández-Lobato et al., 2016a).

- **Pareto Efficient Global Optimization** (ParEGO) (Knowles, 2006) is a unique approach that reduces the multidimensional problem to a single dimension. In each iteration, a weight vector λ is sampled and a new objective function is generated.

  $$f_\lambda(x) = max_i(\lambda_i f_i(x) + \rho \sum_i \lambda_i f_i(x)$$

  Where ρ is typically chosen to be a small value eg. 0.05. From this point, any single objective acquisition can be used to determine the next point. While ParEgo is a straight-forward approach, it can under-perform when it comes to improvement in hyper-volume (Ponweiser et al., 2008).

- **Expected Hyper-volume Improvement** (EHI) (Emmerich and Klinkenberg, 2008) is the natural extension of EI to the multiobjective setting. For each predicted point, the acquisition value is the expected improvement in hyper-volume over the current Pareto set.

  $$EHI(x) = \int p(y|x,\theta)\left(HV(S \bigcup \{y\}) - HV(S)\right) dy$$

Where S is the current Pareto set, θ are the model parameters and HV denotes the hypervolume function.

The main drawback of the method is that the above integral is very difficult to accurately approximate. Moreover, cost of the naïve approximation that uses a grid of points grows exponentially w.r.t. the number of objective functions.

- **S-metric selection-based efficient global optimization** (SMSego) (Ponweiser et al., 2008) uses an optimistic estimate of the objective functions to speed up the computations. The acquisition value of each point is the improvement in hypervolume by the optimistic estimate.

$$\tilde{y} = \mu_y - c\sigma_y$$

$$SMSego(x) = HV(S\bigcup\{\tilde{y}\}) - HV(S)$$

  Where μ_y and σ_y are the mean and standard-deviation given by the predictor and c is a constant. The advantage of SMSego is that it is quick and simple to calculate, however, the downside is that it cannot cope with non-Gaussian predictions.

- **Pareto Active Learning** (PAL) (Zuluaga et al., 2013) takes SMSego one step further without increasing its computational cost. For each point x, PAL defines a region of uncertainty.

$$Q(x) = \{\tilde{y} | \mu_y - c\sigma_y < \tilde{y} < \mu_y + c\sigma_y\}$$

  This gives a rough estimate for the region where y is located with probability depending on the value of the constant c. Then each point is classified as Pareto-optimal, non-Pareto-optimal or unclassified. Pareto-optimal points are the ones where no points in $Q(x)$ are dominated by the current Pareto-set. The points where the whole region of $Q(x)$ is dominated by the current Pareto-set are classed as non-Pareto-optimal. The rest of the points are unclassed. In each iteration, the next candidate is the unclassed point with the largest uncertainty area (largest volume of $Q(x)$).

  The advantage of this method is that it is cheap and very efficient at reducing the uncertainty near the Pareto-front. However, reduction in uncertainty does not necessarily correlate with increased hypervolume. A further disadvantage is that the method does not cope well with noisy data because it classifies too many points as ‘unclassified’.

- **Sequential uncertainty reduction** (SUR) (Picheny, 2015) is the extension of PI to the multidimensional case. The process is similar to EHI, except that the acquisition value is the probability of improving the hypervolume. SUR suffers from the same problems as EHI. High computational cost which makes it viable only for 2-3 dimensions.

- **Predictive Entropy Search** (PESMO) (Hernández-Lobato et al., 2016a) is the adaptation of entropy search to the multiobjective domain. Its acquisition value is the expected reduction in the entropy of the Pareto-set.

  $$PESMO(x) = H(S) - E_y(H(S \bigcup y))$$

  While it has been shown that PESMO often outperforms the other approaches, it suffers from the same issue as EHI. The expected value cannot be directly calculated and can be expensive to approximate.

Our choice of data-collection strategy was SMSego which we could justify by its straightforward implementation and reasonable performance. While it is important to employ an efficient acquisition function, the focus of the project was to evaluate DGPs and Joint DGPs in a multiobjective optimization setting. The choice of acquisition function did not play a crucial role in this regard.

Fortunately, GPs and DGPs are not affected by SMSego's inability to deal with non-Gaussian predictions, because the predicted distribution is always Gaussian for these two models.

2.2 Gaussian Processes

This section presents a review of Gaussian Process models. It talks about the underlying assumptions and mechanisms that define a GP, followed by the introduction of sparse approximations that are used in order to reduce the high computational cost of GPs. Finally, the section analyses their role in the context of Bayesian Optimization.

2.2.1 Description of GPs

A Gaussian Process (GP) (Williams and Rasmussen, 1996) is a distribution over functions that can be used for Bayesian regression. $f : \mathcal{X} \to \mathcal{R}$ where f is an infinite dimensional mapping from the input space $\mathcal{X}$ to the real numbers $\mathcal{R}$:

$$p(f|D) = \frac{p(f)p(D|f)}{p(D)}$$

The distributions defined by GPs share the property that for any finite subset of the input space $\{x_1, \dots, x_n\}$, the marginal probability $p(f(x_1), \dots, f(x_n))$ is a multivariate Gaussian distribution.

This property can be used to calculate the marginal likelihood to make predictions.

Hyperparameters

The hyperparameters of a Gaussian Process are the mean function μ and the covariance function K:

$$p(f(\boldsymbol{x})) = \mathcal{N}(\boldsymbol{\mu}, \Sigma)$$

$$\boldsymbol{\mu} = [\mu(x_1), \dots, \mu(x_n)]^T$$

$$\Sigma = \begin{bmatrix} K(x_1,x_1) & K(x_1,x_2) & \dots & K(x_1,x_n) \\ K(x_2,x_1) & K(x_2,x_2) & \dots & K(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ K(x_n,x_1) & K(x_n,x_2) & \dots & K(x_n,x_n) \end{bmatrix}$$

In most use-cases, including this project, the mean μ is fixed at 0 since we have no prior knowledge of the function.

The covariance function K controls the ‘wiggliness’ of the function. It determines how much each datapoint correlates with the others. It can be any function as long as Σ is a positive definite matrix for all $\boldsymbol{x}$, ie. a valid covariance matrix. In this project, we used two types: squared exponential and Matérn.

The squared exponential kernel is a straight forward kernel function that is widely used across different domains due to its simple form and straight-forward implementation.

$$K_{se}(x,x') = \sigma_f^2 exp\left(-\left(\frac{x-x'}{l}\right)^T\left(\frac{x-x'}{l}\right)\right) + \delta(x,x')\sigma_n^2$$

The first term is exponential of the length-scale l adjusted square distance. The lengths-scale is usually kept separate for each input dimension in order to automatically determine how relevant they are (Automatic Relevance Determination or ARD). If the length-scale is large for a given input dimension then it will only have a minor impact on the outcome. The second term accounts for the measurement noise in the samples. Furthermore, the addition of σ_n^2 to the diagonal of Σ ensures the computational stability of the calculations.

The second kernel that we used is the Matérn kernel. The reason that we used it is that Spearmint[1], a toolkit for Bayesian Optimization, employs a Gaussian Process with Matérn covariance function as its underlying model. Spearmint is an acknowledged tool both in academia an industry and therefore serves as a good baseline for our experiments.

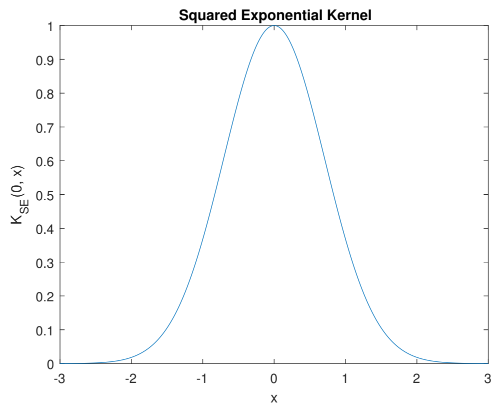
The Matérn kernel that is used by Spearmint is constructed as follows.

$$K_{Mat}(x,x') = \sigma_f^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\left|\frac{x-x'}{l}\right|\right)^{\nu} K_{\nu}\left(\sqrt{2\nu}\left|\frac{x-x'}{l}\right|\right) + \delta(x,x')\sigma_n^2$$

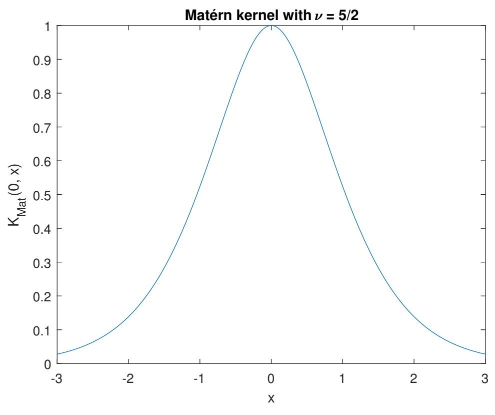
Where ν is a constant parameter and K_ν is the modified Bessel function (Jin and Jjie, 1996). At $\nu \to \infty$, the function is equivalent to the squared exponential kernel. Typical values for ν are $3/2$ and $5/2$. Spearmint uses $\nu = 5/2$. In this case, the function takes the following form:

$$K_{Mat}(x,x') = \sigma_f^2 \left(1 + \sqrt{5}\left|\frac{x-x'}{l}\right| + \frac{5}{3}\left|\frac{x-x'}{l}\right|^2\right) exp\left(-\left|\sqrt{5}\frac{x-x'}{l}\right|\right) + \delta(x,x')\sigma_n^2$$

The advantage of the Matérn covariance function is that it does not converge to 0 as rapidly as the squared exponential function does (Figure 2.2a and 2.2b). This allows distant data-points to influence the predictions while the impact of nearby data-points remains the same.



(a) Squared Exponential covariance function



(b) Matérn covariance function

Fig. 2.2 Comparison of covariance functions in the one dimensional case.

[1]https://github.com/HIPS/Spearmint

The practical issue of covariance functions is that it is difficult to choose one that is right for the problem. It is almost always the case that a cleverly handcrafted covariance function performs better than a general one, however, constructing one requires prior knowledge of the behavior of the function. DGPs offer a solution to this problem by remapping the inputs in the hidden layers which makes them less sensitive to the choice of covariance function.

Optimizing the hyperparameters

One approach to optimizing the hyperparameters is Maximum Likelihood (ML) training. As the name suggests, the hyperparameters are set to maximize the marginal likelihood of the observations.

$$\theta = argmax_\theta p(\boldsymbol{y}|\boldsymbol{x},\theta)$$

$$log\left(p(\boldsymbol{y}|\boldsymbol{x},\theta)\right) = -\frac{1}{2}\boldsymbol{y}^T\Sigma^{-1}\boldsymbol{y} - \frac{1}{2}log|\Sigma| - \frac{n}{2}log2\pi$$

The log-likelihood was derived by taking the logarithm of the Gaussian probability density function.

While ML training lends itself when we have no prior knowledge of the function, it can sometimes be an issue that the length-scale l and noise σ_n take extreme values and the computation becomes unstable.

The second commonly used approach is Maximum-a-Posteriori (MAP) training. Here, the hyperparameters have a prior distribution and the goal is to maximize the posterior likelihood.

$$\theta = argmax_\theta p(\boldsymbol{y}|\boldsymbol{x},\theta)p(\theta)$$

In the presence of prior distributions, MAP outperforms ML training.

Regression

From the assumption that the data can be fully explained as a sample from a multivariate Gaussian distribution, it follows that we can make predictions $\boldsymbol{y}'$ at any finite set of inputs $\boldsymbol{x}' = [x'_1, \ldots, x'_k]^T$.

$$\begin{bmatrix}\boldsymbol{y}\\ \boldsymbol{y}'\end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix}K(\boldsymbol{x},\boldsymbol{x}) & K(\boldsymbol{x},\boldsymbol{x}')\\ K(\boldsymbol{x}',\boldsymbol{x}) & K(\boldsymbol{x}',\boldsymbol{x}')\end{bmatrix}\right)$$

Where

$$K(\boldsymbol{x}',\boldsymbol{x}) = K(\boldsymbol{x},\boldsymbol{x}')^T = \begin{bmatrix} K(x_1',x_1) & K(x_1',x_2) & \dots & K(x_1',x_n) \\ K(x_2',x_1) & K(x_2',x_2) & \dots & K(x_2',x_n) \\ \vdots & \vdots & \ddots & \vdots \\ K(x_n',x_1) & K(x_n',x_2) & \dots & K(x_n',x_n) \end{bmatrix}$$

$$K(\boldsymbol{x}',\boldsymbol{x}') = \begin{bmatrix} K(x_1',x_1') & K(x_1',x_2') & \dots & K(x_1',x_n') \\ K(x_2',x_1') & K(x_2',x_2') & \dots & K(x_2',x_n') \\ \vdots & \vdots & \ddots & \vdots \\ K(x_n',x_1') & K(x_n',x_2') & \dots & K(x_n',x_n') \end{bmatrix}$$

The distribution of the predictions is always a multivariate Gaussian. In practice, we always make independent predictions which means that $\boldsymbol{x}'$ only includes a single value. As a result, each prediction has a scalar value for variance as opposed to a full covariance matrix.

The prediction takes the following form. The derivation is included in Appendix A.

$$\boldsymbol{y}'|\boldsymbol{y} \sim \mathcal{N}\left(K(\boldsymbol{x}',\boldsymbol{x})K(\boldsymbol{x},\boldsymbol{x})^{-1}\boldsymbol{y}, K(\boldsymbol{x}',\boldsymbol{x}') - K(\boldsymbol{x}',\boldsymbol{x})K(\boldsymbol{x},\boldsymbol{x})^{-1}K(\boldsymbol{x},\boldsymbol{x}')\right)$$

The computational cost of making calculating the marginal likelihood is $O(N^3)$ in time and $O(N^2)$ in space since it is dominated by the cost of inverting $K(\boldsymbol{x},\boldsymbol{x}) = \Sigma$. The cost of making K predictions is $O(KN^2 + K^2N)$ in time and $O(N^2 + K^2)$ in space which is dominated by the cost of calculating the covariance matrix for the predictions.

2.2.2 Sparse Gaussian Processes

This section gives an overview of sparse GPs. For the use of sparse approximations in the project, refer to Section 3.2.1. For a more in-depth discussion on spare GPs, see (Snelson and Ghahramani, 2006).

As mentioned earlier, one of the disadvantages of GPs is their high training and prediction costs: $O(N^3)$ and $O(N^2)$ respectively. To mitigate the issue, sparse approximations (Snelson and Ghahramani, 2006) can be utilized. Using $M \ll N$ pseudo inputs, the costs can be reduced to $O(M^2N)$ and $O(M^2)$ respectively.

With this setup, the pseudo targets play the role of latent function values. Denote the pseudo inputs and pseudo targets with $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{f}})$. We can derive the distribution for a finite set of inputs x' analogously to GPs.

$$\boldsymbol{y}'|\bar{\boldsymbol{f}} \sim \mathcal{N}\left(K(\boldsymbol{x}',\bar{\boldsymbol{x}})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}\bar{\boldsymbol{f}}, K(\boldsymbol{x}',\boldsymbol{x}') - K(\boldsymbol{x}',\bar{\boldsymbol{x}})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}K(\bar{\boldsymbol{x}},\boldsymbol{x}')\right)$$

For a single input x' we get:

$$y'|\bar{\boldsymbol{f}} \sim \mathcal{N}\left(K(x',\bar{\boldsymbol{x}})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}\bar{\boldsymbol{f}}, K(x',x') - K(x',\bar{\boldsymbol{x}})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}K(\bar{\boldsymbol{x}},x')\right)$$

Where $K(x',\bar{\boldsymbol{x}}) = K(\bar{\boldsymbol{x}},x')^T = [K(x',\bar{\boldsymbol{x}}_1),\dots,K(x',\bar{\boldsymbol{x}}_1)_M]$.

Assuming that the outputs are generated i.i.d. for the inputs, we can get the likelihood of the data.

$$p(\boldsymbol{y}|\boldsymbol{x},\bar{\boldsymbol{x}},\bar{\boldsymbol{f}}) = \prod_n p(y_n|x_n,\bar{\boldsymbol{x}},\bar{\boldsymbol{f}}) = \mathcal{N}\left(K(\boldsymbol{x},\bar{\boldsymbol{x}})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}\bar{\boldsymbol{f}},\Lambda\right)$$

Where $\Lambda = diag(\lambda_n)$, $\lambda_n = K(x_n,x_n) - K(x_n,\bar{\boldsymbol{x}})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}K(\bar{\boldsymbol{x}},x_n)$.

A naïve approach would train $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{f}}$ to maximize the likelihood of the training data. This would be problematic due to the large number of parameters a likely lead to overfitting. We can get much more accurate results by integrating out the pseudo targets $\bar{\boldsymbol{f}}$.

The prior distribution of $\bar{\boldsymbol{f}}$ is a Gaussian prior with the covariance matrix being defined by the kernel function of the GP. This is a reasonable prior because one can assume that the pseudo targets are distributed similarly to the real data.

$$\bar{\boldsymbol{f}} \sim \mathcal{N}(0, K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}}))$$

The posterior over the pseudo targets $\bar{\boldsymbol{f}}$ can be calculated analytically using Bayes rule.

$$p(\bar{\boldsymbol{f}}|\boldsymbol{y},\boldsymbol{x},\bar{\boldsymbol{x}}) = \mathcal{N}\left(K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})Q^{-1}K(\bar{\boldsymbol{x}},\boldsymbol{x})\Lambda^{-1}\boldsymbol{y}, K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})Q^{-1}K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})\right)$$

Where $Q = K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}}) + K(\bar{\boldsymbol{x}},\boldsymbol{x})\Lambda^{-1}K(\boldsymbol{x},\bar{\boldsymbol{x}})$.

In order to obtain a predictive distribution, we have to integrate out the pseudo targets with the posterior.

$$p(y'|x',\boldsymbol{y},\boldsymbol{x},\bar{\boldsymbol{x}}) = \int p(y'|x',\bar{\boldsymbol{x}},\bar{\boldsymbol{f}})p(\bar{\boldsymbol{f}}|\bar{\boldsymbol{x}})d\bar{\boldsymbol{f}} = \mathcal{N}\left(\mu',\sigma'^2\right)$$

$$\mu' = K(x',\bar{\boldsymbol{x}})Q^{-1}K(\bar{\boldsymbol{x}},\boldsymbol{x})\Lambda^{-1}\boldsymbol{y}$$

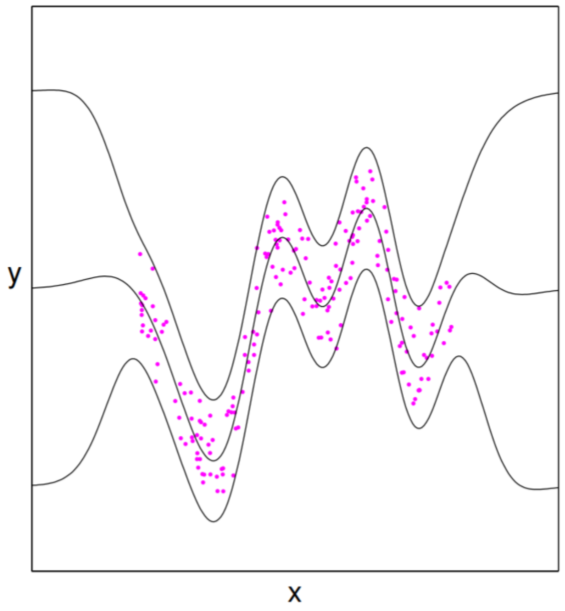
$$\sigma'^2 = K(x',x') - K(x',\bar{\boldsymbol{x}})(K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1} - Q^{-1})K(\bar{\boldsymbol{x}},x')$$

The pseudo inputs $\bar{\boldsymbol{x}}$ can be trained using stochastic gradient ascent along with the hyperparameters using the marginal likelihood.

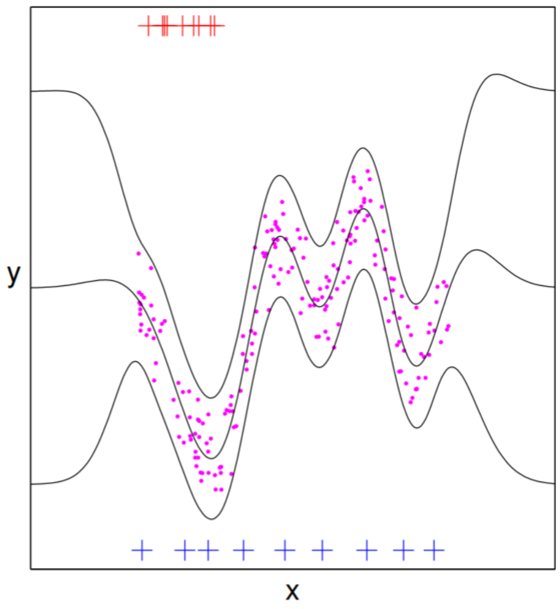
$$p(\boldsymbol{y}|\boldsymbol{x},\bar{\boldsymbol{x}},\theta) = \int p(\boldsymbol{y}|\boldsymbol{x},\bar{\boldsymbol{x}},\bar{\boldsymbol{f}})p(\bar{\boldsymbol{f}}|\bar{\boldsymbol{x}})d\bar{\boldsymbol{f}} = \mathcal{N}\left(0, K(\boldsymbol{x})K(\bar{\boldsymbol{x}},\bar{\boldsymbol{x}})^{-1}K(\bar{\boldsymbol{x}},\boldsymbol{x}) + \Lambda\right)$$

Since Λ is diagonal, the cost is dominated by the matrix multiplication $K(\bar{\boldsymbol{x}},\boldsymbol{x})\Lambda^{-1}K(\boldsymbol{x},\bar{\boldsymbol{x}})$ in the calculation of Q which has cost $O(M^2N)$. The predictions can be obtained in $O(M)$ time for the mean and $O(M^2)$ for the variance.

Figure 2.3a shows the full GP model and Figure 2.3b shows the sparse model obtained from gradient ascent training.



(a) Full GP model.



(b) Sparse GP model.

Fig. 2.3 Comparison of full GP and sparse GP models. The initial positions of the pseudo inputs are denoted by red crosses and the positions of the pseudo inputs after stochastic gradient ascent training are denoted by blue crosses. [2]

One concern is that the pseudo inputs might overfit the training data. However, even in the case when $M = N$ i.e. there is a pseudo point for each training point ($\boldsymbol{x} = \bar{\boldsymbol{x}}$), the equations simply revert to a full GP model.

2.2.3 Gaussian Processes in the context of Bayesian Optimization

In Bayesian Optimization, it is essential to have a statistical model that provides high-quality predictions with uncertainty estimates. All of the data collection strategies rely on these predictions to make the best possible decision about the next evaluation.

Not all predictive models perform well in an optimization setting. First and foremost, the model must be able to give sensible predictions and uncertainties even when the data is scarce. Typically, the high cost of evaluation limits the process to 10-1000 evaluations. Some

[2]Image source: (Snelson and Ghahramani, 2006)

models, such as deep neural networks, perform poorly in this environment which makes them a suboptimal choice.

However, there are trends in optimization problems that we can take advantage of. The data often has low volume and dimensionality. This advocates employing Gaussian Processes, since they work very well with limited, low-dimensional data and their high computational cost is dwarfed by the cost of evaluating the objective function.

The two advantages of Gaussian Processes are that they can work from very few data-points and that they are flexible. They are also very good at coping with noisy observations.

Gaussian Processes are non-parametric models since they are fully defined by the set of hyper-parameters $\theta = \{l, \sigma_f, \sigma_n\}$. The underlying assumption of GPs is that the samples are drawn from an infinite dimensional function therefore the complexity of the model does not increase with more data.

However, there are some non-trivial difficulties that need to be overcome when selecting the hyper-parameters. Firstly, the covariance function has a large impact on the performance of the model and it is difficult to choose without any prior knowledge of the objective function. One approach is the use a kernel that performs well under most tasks, such as the squared exponential kernel with a separate length-scale for each dimension but such an approach cannot compete with a carefully handcrafted covariance function. Secondly, the hyper-parameters need to be tuned. This is done in one of two ways. Maximum Likelihood (ML) training maximizes the marginal likelihood. It is straight-forward, but if we have any prior knowledge, Maximum-a-Posteriori (MAP) training works by maximizing the likelihood of the posterior.

A second drawback of GPs is their high computational cost. The cost of calculating the inverse covariance matrix is $O(N^3)$ in time and $O(N^2)$ in space where N is the number of training samples. This entails that GPs are not a viable choice when the number of training samples is large, however, this is not an issue in Bayesian Optimization.

GPs are widespread in Bayesian Optimization both in academia and in the industry. Tools such as Spearmint rely on GPs as their predictive model.

2.3 Deep Gaussian Processes

Deep Gaussian Processes (Damianou and Lawrence, 2013) are multilayer generalizations of Gaussian Processes. Since GPs are equivalent to infinitely wide neural networks, DGPs are equivalent to infinitely wide, multi-layer neural networks. The inclusion of hidden layers makes DGPs more general and flexible predictors than GPs.

Unlike Section 2.2, this section does not give an in-depth description of DGPs. Instead, we summarize the historical work and recent accomplishments in the area. In Section 3.2, we include detailed information on the exact use of DGPs in the project.

The section is concluded with a discussion about the advantages and disadvantages of using DGPs in Bayesian Optimization.

2.3.1 Brief description of DGPs

This section summarizes the DGP model presented in (Damianou and Lawrence, 2013). While we touch on all the key points in the paper, we highly recommend reading it for a more in-depth discussion.

To simplify the explanation, we are describing a DGP with only a single hidden layer and one node per layer. The architecture is shown on Figure 2.4. Moreover, follow suit with the original paper and introduce DGPs in an unsupervised learning scenario.

In this setup, Z is the unobserved parent node and the observed outputs are $\boldsymbol{y}$. The outputs of the layers are defined as $f^X \sim \mathcal{GP}(0, K_X(\boldsymbol{z},\boldsymbol{z}))$ and $f^Y \sim \mathcal{GP}(0, K_Y(\boldsymbol{x},\boldsymbol{x}))$ where K_X and K_Y are the kernel functions of the layers analogous to a GP.

The inputs to the next layer are simply the outputs of the previous one with added Gaussian noise. Figure 2.5 serves as a demonstration using a single hidden layer that contains two nodes.

$$\boldsymbol{x} = f^X(\boldsymbol{z}) + \varepsilon_X \quad \varepsilon_X \sim \mathcal{N}(\boldsymbol{0}, \sigma_X^2 \boldsymbol{I})$$

$$\boldsymbol{y} = f^Y(\boldsymbol{x}) + \varepsilon_Y \quad \varepsilon_Y \sim \mathcal{N}(\boldsymbol{0}, \sigma_Y^2 \boldsymbol{I})$$

This structure can be naturally extend by adding more hidden layers (vertical extension) or by adding more nodes into the already existing layers (horizontal extension) and fully connecting them to the layer below and above. The architecture of the Joint DGP model that we used in the project is shown in Figure 3.3.

Training is still unclear. One can see that the number of parameters that need to be trained grew significantly ($\boldsymbol{x}$). As it turns out, we can marginalize the whole latent space out in a Bayesian manner. The Bayesian training does not only drastically reduce the number of parameters, but also enables an automatic Occam's razor to deal with potential overfitting problems. In the next section, we present a variational approach that obtains a lower bound to the marginal likelihood.

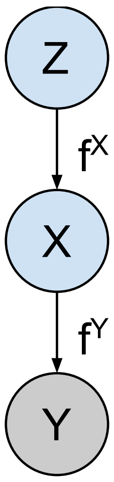


Fig. 2.4 Simplified view of DGPs.

2.3.2 Approximate inference

This section gives the example of variational marginalization of the latent variables, but as we show in the next section, there are other options for approximate inference.

During training, we seek to maximize the marginal likelihood.

$$logp(\boldsymbol{y}) = log \int p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{xy}$$

Unfortunately, this integral is intractable due to the non-linear way in which $\boldsymbol{x}$ and $\boldsymbol{z}$ interact.

One can obtain a lower-bound to the marginal likelihood $\mathcal{F} \leq logp(\boldsymbol{y})$ using Jensen's inequality.

$$\mathcal{F} = \int \mathcal{Q} log \frac{p(\boldsymbol{y}, \boldsymbol{f}^Y, \boldsymbol{x}, \boldsymbol{f}^X, \boldsymbol{z})}{\mathcal{Q}}$$

The role of $\mathcal{Q}$ will be explained later.

The joint distribution can be expanded.

$$p(\boldsymbol{y}, \boldsymbol{f}^Y, \boldsymbol{x}, \boldsymbol{f}^X, \boldsymbol{z}) = p(\boldsymbol{y}|\boldsymbol{f}^Y)p(\boldsymbol{f}^Y|\boldsymbol{x})p(\boldsymbol{x}|\boldsymbol{f}^X)p(\boldsymbol{f}^X|\boldsymbol{z})p(\boldsymbol{z})$$

The integral is still intractable. The difficult terms are $p(\boldsymbol{f}^Y|\boldsymbol{x})$ and $p(\boldsymbol{f}^X|\boldsymbol{z})$ due to the non-linear relationship they have with $\boldsymbol{x}$ and $\boldsymbol{z}$.

Tractability can be achieved through a two-step process.

First, we can use the key result of (Titsias and Lawrence, 2010) which is that with the introduction of pseudo input-target pairs, the prior distributions can be propagated through the non-linear mapping f. This is analogous to Sparse GPs. For each mapping, we augment the probability space with M pseudo inputs (also called inducing points) $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{z}}$ and we marginalize the pseudo targets $\boldsymbol{u}^X$ and $\boldsymbol{u}^Y$.

Second, we can cleverly define $\mathcal{Q}$ to cancel out the difficult terms $p(\boldsymbol{f}^Y|\boldsymbol{x})$ and $p(\boldsymbol{f}^X|\boldsymbol{z})$.

$$\mathcal{Q} = p(\boldsymbol{f}^Y|\boldsymbol{u}^Y,\boldsymbol{x})q(\boldsymbol{u}^Y)q(\boldsymbol{x})p(\boldsymbol{f}^X|\boldsymbol{u}^X,\boldsymbol{z})q(\boldsymbol{u}^X)q(\boldsymbol{z})$$

Where $q(\boldsymbol{u}^Y)$ and $q(\boldsymbol{u}^X)$ are free-form variational distributions and $q(\boldsymbol{x})$ and $q(\boldsymbol{z})$ are factorized Gaussians w.r.t the dimensions.

$$q(\boldsymbol{x}) = \prod_q^Q \mathcal{N}(\mu_q^X, S_q^X) \quad q(\boldsymbol{z}) = \prod_q^{Q_Z} \mathcal{N}(\mu_q^Z, S_q^Z)$$

Finally, we arrive to a tractable lower bound.

$$\mathcal{F} = g_Y + r_X + \mathcal{H}_q(\boldsymbol{x}) - KL(q(\boldsymbol{z})||p(\boldsymbol{z}))$$

$$g_Y = E_{p(\boldsymbol{f}^Y|\boldsymbol{u}^Y,\boldsymbol{x})q(\boldsymbol{u}^Y)q(\boldsymbol{x})}\left(logp(\boldsymbol{y}|\boldsymbol{f}^Y) + log\frac{p(\boldsymbol{u}^Y)}{q(\boldsymbol{u}^Y)}\right)$$

$$r_X = E_{p(\boldsymbol{f}^X|\boldsymbol{u}^X,\boldsymbol{z})q(\boldsymbol{u}^X)q(\boldsymbol{z})}\left(logp\boldsymbol{x}|\boldsymbol{f}^X) + log\frac{p(\boldsymbol{u}^X)}{q(\boldsymbol{u}^X)}\right)$$

Where $\mathcal{H}$ denotes the entropy of the distribution and KL refers to the Kullback-Leibler divergence. For a more detailed, step-by-step derivation see (Damianou and Lawrence, 2013).

The training of the DGP models involves optimizing the above expression w.r.t. the hyperparameters for each GP mapping ($\theta = \{\sigma_f, \sigma_n, l\}$), the inducing points ($\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{z}}$) and the variational parameters.

The issue with the variational approach is that the number of variational parameters still grow linearly with the size of the data. As a result, DGPs are notoriously difficult to train because they can get stuck in a local optima. The next section offers multiple solutions to this problem.

2.3.3 Recent works

This section outlines three recent and remarkable results in the area of approximate inference for DGPs. They all enable scaling of DGPs to medium-large datasets. This helps to place our work in the context of ongoing research.

Firstly, a paper by (Bui et al., 2016), in which Approximate Expectation Propagation is used for approximate inference. Our implementation of the DGP model is based on this paper.

Secondly, a paper by (Dai et al., 2015), in which they introduce a recognition model: Variational Auto-Encoded deep Gaussian process (VAE-DGP).

Thirdly, a paper by (Salimbeni and Deisenroth, 2017), in which they present a doubly stochastic variational inference algorithm, which does not force independence between layers.

Deep Gaussian Processes for Regression using Approximate Expectation Propagation

In this paper, Expectation Propagation (Minka, 2001) is used for approximate inference with the goal in mind to scale DGPs up to be able to cope with large quantities of training data as well as to apply them to regression tasks.

The paper demonstrates these qualities by applying DGP models to large datasets and shows that they always outperform GPs. An in-depth description of the content of the paper is given in Section 3.2.

When we applied this model in the Multiobjective Optimization setting, one concern we had was that the model might underperform since it was not designed for small datasets. While we could not exclude the possibility that a different approximate inference method might perform better than EP, we did find that the log-likelihood of the DGP models were always higher than the baseline GP models.

Variational Auto-Encoded Deep Gaussian Processes

In order to scale up DGPs to handle large datasets, the authors augment DGPs with a variationally auto-encoded inference mechanisms. Said mechanisms are referred to as recognition models. A recognition model can be used to constrain the variational posterior distributions of latent variables. In turn, this lead to reduction in the number of parameters for optimization since the number of variational parameters do not grow linearly with the size of the data.

The second result of the paper is that it establishes the auto-encoded variational lower bound, which can be computed in a distributed manner. This enables further scaling of the models, although it does not lead to reduced costs in the computation.

Doubly Stochastic Variational Inference for Deep Gaussian Processes

A noteworthy result by (Salimbeni and Deisenroth, 2017) is that the authors lift the forced assumption of independence between the layers in order to build more flexible models. The authors introduced a doubly stochastic variational inference algorithm to cope with the intractable posterior distribution, which in addition, enabled scaling to large datasets.

The first source of stochasticity is the sampling approach that is used to approximate the marginals. Samples are drawn from the variational posterior formed by the inducing points. This is enabled by the fact that the marginals conditioned on the previous layer only depend on the corresponding inputs.

The second source of stochasticity comes from mini-batch training. This step is required for the scalability of any model.

2.3.4 Deep Gaussian Processes in the context of Bayesian Optimization

Similarly to GPs, DGPs are flexible, non-parametric models. They can handle noisy and low volumes of training data and still make sensible predictions. Moreover, DGPs offer a solution to a key issue with GPs which is the problem of selecting appropriate covariance functions and determining the hyper-parameters. DGPs are not sensitive to the choice of covariance function due to the fact that the hidden layers can warp the inputs and reduce or extend dimensionality resulting in an automatically designed kernel that is flexible and fits the problem.

Unlike GPs, DGPs have not been extensively applied to Bayesian Optimization problems due to their history of training difficulties. However, with the recent results in the field, the became promising candidates to be used as predictive models.

Computational cost

Ordinary, the computational cost of DGPs would be $O(LN^3)$, however, this can be reduced to $O(LNM^2)$ (where M is the number of inducing points per node) by using sparse GP approximations. One implication of this is that the cost grows quadratically with the number of inducing points but only linearly with the number of layers, therefore, the errors introduced by the sparse approximation can be mitigated by increasing the depth and width of the network.

2.4 Bayesian Optimization in hardware design

Designing hardware is one of the many applications of Bayesian Optimization in the industry. It is an appealing problem to solve, since the testing of configurations is often very expensive and the number of parameters is too high for a human expert to tune intelligently.

Black-box Bayesian Optimization has been shown to outperform other approaches both in the number of samples required and the quality of the solution found. (Orianna DeMasi, 2014) demonstrated the effectiveness of BO in micro architectural processor design. A straight forward approach, using GP models with EI as the acquisition function was able to solve the design problem in as few as 15-20 evaluations.

A more recent paper (Hernández-Lobato et al., 2016c) examined the design of neural network hardware accelerators, the same problem that appears in this project. The paper considered the idea of using decoupled evaluations in multiobjective optimization in order to reduce testing costs. Its significant difference from our project is that the authors focused on refining the optimization process whereas our work aims to improve the predictive model.

2.4.1 Neural Network Hardware Accelerators

The real-world problem that this project tackles is the problem of designing neural network hardware accelerators. Despite that we are treating the design process as a multiobjective black-box optimization problem, it is important to outline the underling design in order to be able to assess the impact of our work.

Deep Neural Networks (DNN) have risen in popularity in the past decade and continue to rise due to their flexibility and power in supervised regression and classification tasks. They are the state-of-the-art technology in areas such as Image Recognition, Speech recognition, Speech Synthesis etc.. However, one issue with DNNs is that they require significant computational resources such as high-performance CPUs/GPUs to use which presents a major obstacle in their integration to smart phones, mobile and battery powered devices. They are too slow and draw too much power to be used in this setting.

Minerva (Reagen et al., 2016) offers a solution to the problem in hardware. Specialized DNN accelerators that both maintain high accuracy and low power consumption. It claims to achieve a 8.1x reduction in power consumption across five datasets without significantly compromising the prediction accuracy. It achieves the results in five stages that are briefly described below:

- Stage 1: Training Space Exploration. Establishing baselines for state-of-the-art ML tasks and selecting a suitable network topology.

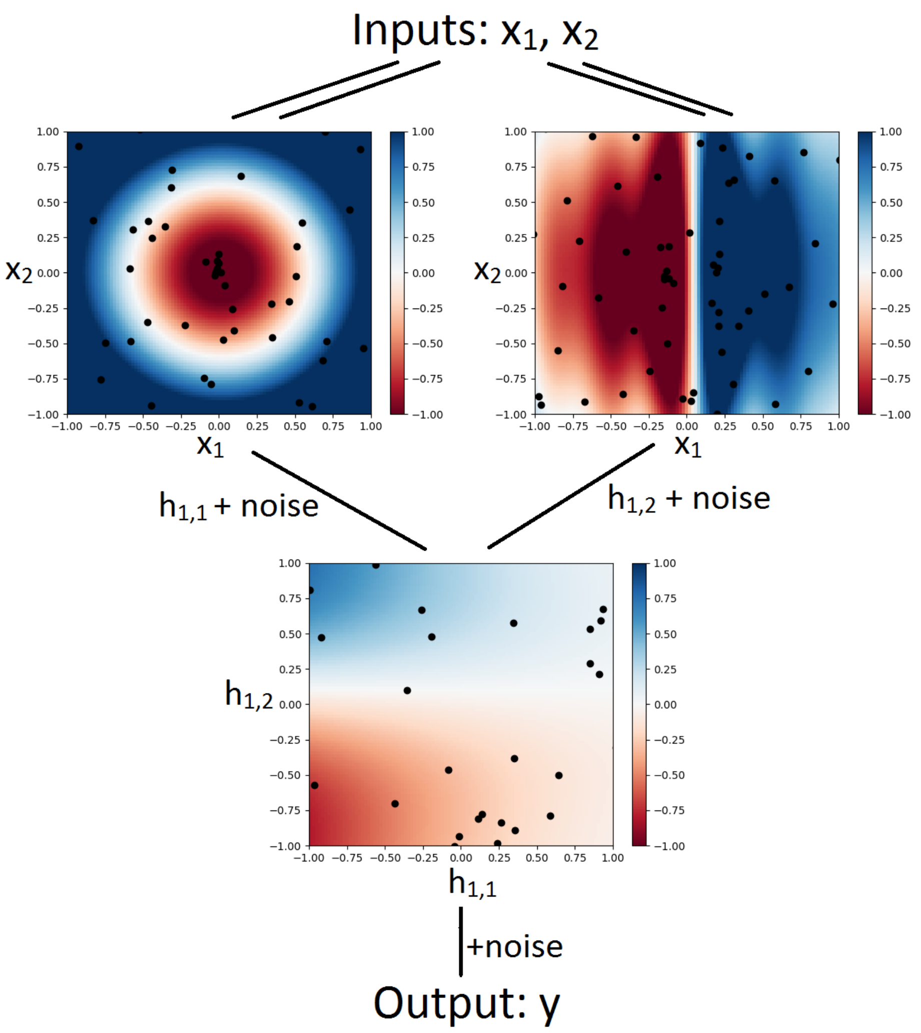
- Stage 2: Microarchitecture Design Space. Selecting adequate microarchitectural parameters such and clock frequency, memory bandwidth
- Stage 3: Data Type Quantization. Restricting the number of bits that are stored in each layer. When a value does not require the full, 16-bits fixed point precision, fewer bits can be used to save computation and bandwidth.
- Stage 4: Selective Operation Pruning. Removes the nodes from the network whose output is predominantly zero.
- Stage 5: SRAM Fault Mitigation. Due to the algorithmic redundancy of DNNs, they have high tolerance for single value faults. This gives room to reduce the SRAM voltage, which in turn increases the number of read-faults, which are then mitigated by rounding the faulty weights towards zero.

The issue with the process is that each stage involves multiple parameters that need to be tuned (Table 2.1). Moreover, testing configurations is expensive both in time and computational resources. They are each tested by emulating the hardware in software and training and testing DNNs on five large datasets. The tool used for testing the configurations is Aladdin (Shao et al., 2014). It can predict the power-performance characteristics of a hardware accelerator within a small error range without having to physically produce and test the configuration.

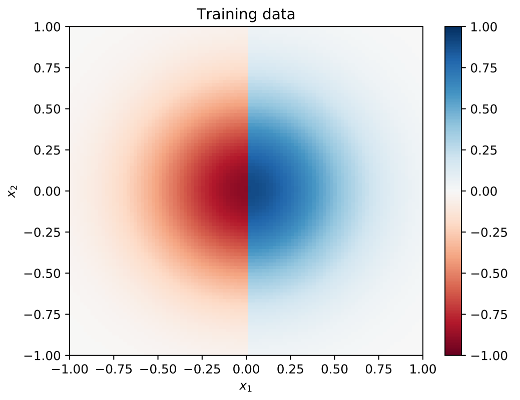
Parameter	Min	Max	Step
Neurons per layer	50	250	1
Learning rate	0.001	1	ε
Dropout rate	0	0.4	ε
L2 penalty	0	0.1	ε
Memory bandwidth	1	32	2^x
Loop parallelism	1	32	2^x
Total number of bits	1	32	2^x
Fraction integer bits	0	1	ε

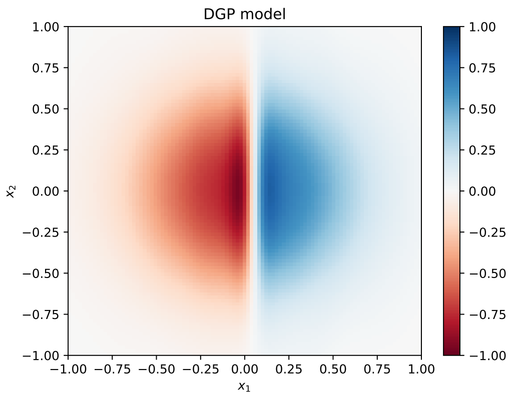
Table 2.1 Tunable parameters of the design

Multiobjective Optimization allows us to find the set of optimal parameters in the power and error tasks with a limited number of evaluations using Aladdin. This does not only significantly reduce the cost of the design process but can also lead to more optimal configurations.



(a) Mappings of the nodes of the DGP. The outputs of each node are colour coded. The locations of the inducing points are marked by black.





(b) Training data.

(c) DGP model.

Fig. 2.5 Demonstration of the DGP model on a 2D toy problem.

Chapter 3

Methods

This chapter contains an in-depth review of SMSego and DGPs for regression followed by the novel extension to DGPs: Joint DGPs. We focus on providing granular, implementation level detail.

Finally, the chapter is concluded with details on the implementation in relation to our experiments.

3.1 S-Metric Selection-based Efficient Global Optimization

S-Metric Selection-based Efficient Global Optimization (SMSego) is the multiobjective optimization approach that we used in this project. This section is a review of (Ponweiser et al., 2008).

As described in the introduction, SMSego is a method for multiobjective optimization that uses an optimistic estimate of the hypervolume improvement as the acquisition function. Its use in the project is justified by its straight-forward implementation and reasonably good performance compared to other optimization methods.

The acquisition value is calculated based on the Lower Confidence Bound (LCB).

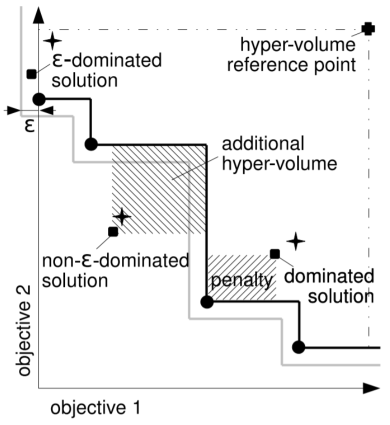
$$y_{pot} = \mu_y - \alpha\sigma_y$$

Where α corresponds to a given confidence level $p_\alpha = (1 - 2\Phi(\alpha))^m$ assuming m objective functions. We found that $\alpha = 2.0$ works well which corresponds to $p_\alpha = 0.002$.

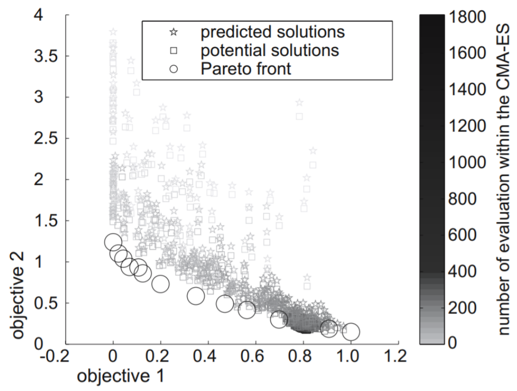
To avoid the problem of having too many points in the Pareto-set where is point is only marginally better than the neighboring points, we introduce the concept of ε-domination. A point ε-dominates another if in each objective function, it either dominates or is within ε distance. We want to adapt ε to the current progress. ε should decrease over time, be sensible in the context of the scale of the objective function and to decrease as the number of points in the Pareto approximation gets too high. This way, the low ε values allow for new points in the Pareto approximation, while under high ε values, more potential points are ε-dominated and therefore it is more difficult to extend the Pareto approximation.

$$\varepsilon = \frac{\Delta\Lambda}{|\Lambda|} + cn_{left}$$

Where Λ denotes the current Pareto approximation and $\Delta\Lambda = max(\Lambda) - min(\Lambda)$. This way, the term $\frac{\Delta\Lambda}{|\Lambda|}$ controls the pacing of the Pareto points. $c = 1 - \frac{1}{2^m}$ is the idealized probability of each remaining evaluation to be a Pareto point and n_{left} is the remaining number of evaluations. This formula has proven to limit the maximum number of points in practice (Emmerich and Naujoks, 2004).



(a) Examples of the non-ε-dominated solution, the dominated solution and the ε-dominated solution.



(b) Pareto frontier example with the predicted and potential solutions.

Fig. 3.1 Examples of SMSego acquisition values.[1]

SMSego distinguished three cases (Figure 3.1):

- Non-ε dominated solution. The acquisition value of the potential point is the increase in hypervolume. $\mathcal{S}(\Lambda \bigcup \{y_{pot}\})\mathcal{S}(\Lambda)$ where $\mathcal{S}$ denotes the hypervolume.
- Dominated solution. For each dominating point, a penalty is applied:

$$p = \sum_{y_i \in \Lambda} \begin{cases} -1 + \prod_{j=1}^{m} \left(1 + y_{pot,j} - y_{i,j}\right) & \text{if } y_i \text{ dominates } y_{pot} \\ 0 \end{cases}$$

[1]Image source: (Ponweiser et al., 2008)

This penalty helps to distinguish between the points when there are no non-ε-dominated potential solutions left.

- ε dominated solution. In this case, the penalty is only applied to the dimensions where the potential solution is strictly dominated.

3.2 Deep Gaussian Processes for Regression

Deep Gaussian Processes (Damianou and Lawrence, 2013) are multilayer extensions of GPs where the output of each hidden layer is used as the input for the next. This section contains a review of (Bui et al., 2016) with focus on the algebraic calculations required for a complete implementation.

There are two key advantages to DGPs over GPs. Firstly, they are more general and flexible due to the mapping provided by the hidden layers. While a single GP is equivalent to an infinitely wide, single layer neural network, a DGP is equivalent to an infinitely wide, multilayer neural network. Secondly, they are less sensitive to the choice of the covariance function. The hidden layers allow for stretching and warping the inputs which results in an automatically designed kernel function in the subsequent layers. On Figure 2.5, we can observe the capability of DGPs in terms of dealing with function cliffs.

Formally, for input and output pairs (x_i, y_i) for $i = 1 \dots N$, the operation of a DGP consisting of L hidden layers can be written as follows:

$$\begin{aligned} \boldsymbol{h_1} &= \boldsymbol{x} \\ p(f_l|\theta_l) &= \mathcal{GP}(f_l; \boldsymbol{0}, \boldsymbol{K_l}) \\ p(\boldsymbol{h_l}|f_l, \boldsymbol{h}_{l-1}, \sigma_l) &= \prod_n \mathcal{N}(h_{l,n}; f_l(h_{l-1,n}), \sigma_l^2) \\ p(y|f_l, \boldsymbol{h}_L, \sigma_L) &= \prod_n \mathcal{N}(y_n; f_l(h_{L,n}), \sigma_L^2) \end{aligned} \tag{3.1}$$

At $L = 1$, the model collapses into a single layer, ‘shallow’, GP model. Typically a hidden layer consists of multiple nodes, although we did not indicate this in our notation.

Similarly to GPs, the cost is dominated by the inversion of the covariance matrix $O(LN^3)$. To reduce the computational cost, we employ sparse approximations to the GPs in each layer (Section 3.2.1). Fortunately, the impact of sparse approximations is mitigated by the multi-layer configuration. The hidden layers restore some of the representative power that is taken away by the sparse approximation.

Calculating the marginal likelihood is intractable so approximate inference techniques are required. Section 3.2.2 describes Expectation Propagation, an approach that allows us to establish a lower bound to the marginal likelihood that can be used for training.

3.2.1 Fully Independent Training Conditional Approximation

The Fully Independent Training Conditional (FITC) (Quiñonero-Candela and Rasmussen, 2005; Snelson and Ghahramani, 2006) approximation reduces the cost from $O(LN^3)$ to $O(LNK^2)$ where N is the number of training instances, K in the number of inducing points per node and L is the number of hidden layers.

The sparsity approximation is formed using K inducing input output pairs $(\boldsymbol{z}_l, \boldsymbol{u}_l)$ for $l = 1, \dots, L$ in each node which, in turn, are used to make predictions on the actual data. The inducing points are assumed to be independent. This creates a semi-parametric representation of the original GP model. A graphical representation of the model is included in Figure 3.2.

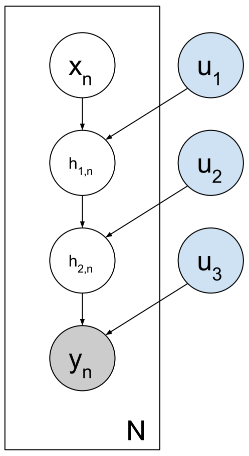


Fig. 3.2 FITC graphical model.

We can formalize the model as follows.

$$
\begin{aligned}
p(\boldsymbol{u}_l|\theta_l) &= \mathcal{N}(0, K(\boldsymbol{u}_l, \boldsymbol{u}_l)) \quad \text{for} \quad l = 1, \dots, L \\
p(h_l|h_{l-1}, u_l, \sigma_l) &= \prod_n \mathcal{N}\left(h_{l,n}; C_{l,n}\boldsymbol{u}_l, R_{l,n}\right) \\
p(y|h_L, u_L, \sigma_L) &= \prod_n \mathcal{N}\left(y_n; C_{L,n}\boldsymbol{u}_L, R_{L,n}\right)
\end{aligned}
\tag{3.2}
$$

where

$$C_{l,n} = K(\boldsymbol{h}_{l-1,n}, \boldsymbol{z}_l)K(\boldsymbol{z}_l, \boldsymbol{z}_l)^{-1}$$

$$R_{l,n} = K(\boldsymbol{h}_{l-1,n}, \boldsymbol{h}_{l-1,n}) - K(\boldsymbol{h}_{l-1,n}, \boldsymbol{z}_l)K(\boldsymbol{z}_l, \boldsymbol{z}_l)^{-1}K(\boldsymbol{h}_{l-1,n}, \boldsymbol{z}_l)^T + \sigma_l \mathcal{I}$$

K denotes the kernel (Section 2.2). The derivation of these results is the same as before (Appendix A).

The locations of the inducing points $\boldsymbol{z}$ are added towards the the set of hyperparameters and are trained jointly. As for $\boldsymbol{u}_l$, we can infer from the training data in with the goal in mind to maximize the marginal likelihood. Section 3.2.2 describes the inference process in detail.

As mentioned earlier, the time complexity of the training is reduced to $O(LNK^2)$ since the covariance matrix that needs to be inverted is only K dimensional. For this to be an improvement over $O(LN^3)$, K needs to be less than N. In our experiments we found that the value $K = 0.1N$ works well in practice. One might argue that fixing K at a constant ratio to N defeats the point of the approximation, since the runtime is technically still $O(N^3)$. The counter-argument is that while the statement is true, fixing K at $0.1N$ is still three orders of magnitude improvement which greatly improves the computational feasibility of the model. In our experiments, $K = 0.1N$ was sufficient, although for large N it might become necessary to establish a non-linear relationship between K and N.

3.2.2 Approximate inference via Expectation propagation

This section describes the inference of the inducing points $\boldsymbol{u} = \{\boldsymbol{u}_l\}_{l=1}^L$ and the training of the model parameters $\alpha = \{\boldsymbol{z}_l, \theta_l\}_{l=1}^L$.

The posterior distribution of the inducing outputs can be written as

$$p(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{u}) \prod_n p(y_n|x_n, \boldsymbol{u}) \tag{3.3}$$

Predictions can be formed using

$$p(y'|x', \boldsymbol{x}, \boldsymbol{y}\alpha) = \int p(y'|x', \boldsymbol{u})p(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{y})d\boldsymbol{u} \tag{3.4}$$

And the marginal likelihood

$$p(\boldsymbol{y}|\alpha) = \int p(\boldsymbol{u}, \boldsymbol{h})p(\boldsymbol{y}|\boldsymbol{u}, \boldsymbol{h}, \alpha)d\boldsymbol{u}d\boldsymbol{h} \tag{3.5}$$

The problem is that neither the posterior distribution $p(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{y})$ nor the marginal likelihood $p(\boldsymbol{y}|\alpha)$ is tractable for multiple layers. The issue is dealt with by using Stochastic Expectation Propagation (SEP) (Li et al., 2015) to approximate both the posterior and the marginal likelihood.

Expectation Propagation

A review of Expectation Propagation (EP) (Minka, 2001) is included in Appendix B.

Expectation Propagation allows us to obtain an approximation to the marginal likelihood by considering the EP energy.

$$\begin{aligned}
log(p(y|\boldsymbol{u})) \approx \mathcal{F}(\alpha) = \phi(\theta) - \phi(\theta_{prior}) + \sum_{n=1}^{N} log\tilde{\mathcal{Z}}_n \\
log\tilde{\mathcal{Z}}_n = log\mathcal{Z}_n + \phi(\theta^{\backslash n}) - \phi(\theta) \\
log\mathcal{Z}_n = log \int q^{\backslash n}(\boldsymbol{u})p(y_n|x_n,\boldsymbol{u})d\boldsymbol{u}
\end{aligned} \tag{3.6}$$

Where θ, $\theta^{\backslash n}$ and θ_{prior} refer to the natural parameters of $q(\boldsymbol{u})$, $q^{\backslash n}(\boldsymbol{u})$ and $p(\boldsymbol{u})$ respectively. $\phi(\theta)$ denotes the log normalizer of the Gaussian distribution with natural parameters θ. These equations were derived from marginalizing and normalizing the posterior (Equation 3.3). For the full derivation, refer to (Seeger, 2005).

EP in its current form is still not efficient enough. Firstly, it is too costly to optimize N data factors in each loop of the optimization process. Secondly, the data-factors take up $O(NK^2)$ space which is clearly not feasible for large N.

Stochastic Expectation Propagation

We can improve on EP by using the Stochastic Expectation Propagation (SEP) (Li et al., 2015) algorithm. In SEP, we make the approximation that all the data factors are the same: $q(\boldsymbol{u}) = p(\boldsymbol{u})g(\boldsymbol{u})^N$. While this might seem drastic at first, we have to realize that $g(\boldsymbol{u})$ is an N dimensional multivariate Gaussian distribution therefore it has sufficient flexibility to approximate the GP posterior. In fact, SEP has been shown to perform close to EP in practice while solving both of the issues presented earlier. The memory requirement of SEP is $O(K^2)$ as opposed to EP's $O(NK^2)$.

A second, advantage of SEP is that it allows for minimization the EP energy directly. In EP, the marginal likelihood is the byproduct of the optimized data-factors, but in SEP, it can be optimized directly. With tied data factors $\tilde{t}_n(\boldsymbol{u}) = g(\boldsymbol{u})$, the cavity distributions $q^{\setminus n}(\boldsymbol{u}) = \frac{q(\boldsymbol{u})}{g(\boldsymbol{u})} = q^{\setminus 1}(\boldsymbol{u})$ are also the same and the EP energy can be simplified.

$$\begin{aligned} log(p(y|\boldsymbol{u})) \approx \mathcal{F}(\alpha) &= \phi(\theta) - \phi(\theta_{prior}) + \sum_{n=1}^{N} \Big(log\mathcal{Z}_n + \phi(\theta^{\setminus 1}) - \phi(\theta) \Big) \\ &= -(N-1)\phi(\theta) + N(\phi(\theta^{\setminus 1}) - \phi(\theta_{prior}) + \sum_{n=1}^{N} log\mathcal{Z}_n \\ log\mathcal{Z}_n &= log \int q^{\setminus 1}(\boldsymbol{u}) p(y_n|x_n, \boldsymbol{u}) d\boldsymbol{u} \end{aligned} \tag{3.7}$$

Direct optimization only works if we can approximate $log\mathcal{Z}_n$. The approximation is presented in the next section. As for the other terms, they can be calculated exactly since $\theta = \theta_{prior} + N\theta_1$ and $\theta^{\setminus 1} = \theta_{prior} + (N-1)\theta_1$ where θ_1 denotes the natural parameters of the tied data factor $g(\boldsymbol{u})$.

The difference between minimizing the data-factors and the EP energy is that minimizing that data-factors is averaging the natural parameters in the term $g(\boldsymbol{u})$ while minimizing the energy averages the moments of $\int q^{\setminus 1}(\boldsymbol{u}) p(y_n|x_n, \boldsymbol{u}) d\boldsymbol{u}$. Note that it is not necessarily the case that optimizing the EP energy and the data-factors lead to the same optima.

Probabilistic backpropagation

Computing $log\mathcal{Z}_n$ is not tractable in a multi-layer setup because the cavity $q^{\setminus 1}$ forms a complex, non-Gaussian distribution in the layers beyond the first one. To tackle this problem, the approximation we are making is moment-matching this non-Gaussian cavity distribution to form a Gaussian one. The process is equivalent to Assumed Density Filtering (Hernández-Lobato and Adams, 2015) under the special case of Gaussian distributions. The approximation to $\mathcal{Z}$ is made in each layer starting from the first one and iteratively progressing towards the output layer.

To make the moment matching approximation, we need to reintroduce the values at the hidden layers $\boldsymbol{h}$. For a two layer model we get:

$$\begin{aligned} \mathcal{Z} &= \int q^{\setminus 1}(\boldsymbol{u}) p(y|x, \boldsymbol{u}) d\boldsymbol{u} \\ &= \int q^{\setminus 1}(\boldsymbol{u}_2) p(y|h_1, \boldsymbol{u}_2) dh_1 d\boldsymbol{u}_2 \int q^{\setminus 1}(\boldsymbol{u}_1) p(h_1|x, \boldsymbol{u}_1) d\boldsymbol{u}_1 \end{aligned} \tag{3.8}$$

Next, we can marginalize out the inducing outputs $\boldsymbol{u}$.

$$\begin{aligned} \mathcal{Z} &= \int q(h_1)q(y|h_1)dh_1 \\ q(h_1) &= \mathcal{N}(m_1, v_1) \\ q(y|h_1) &= \mathcal{N}(m_2|h_1, v_2|h_1) \end{aligned} \tag{3.9}$$

The means and variances take the form :

$$\begin{aligned} m_1 &= K(x,z_1)K(z_1,z_1)^{-1}m_1^{\backslash 1} \\ v_1 &= \sigma_1 + K(x,x) - K(x,z_1)K(z_1,z_1)^{-1}K(z_1,x) + K(x,z_1)K(z_1,z_1)^{-1}V_1^{\backslash 1}K(z_1,z_1)^{-1}K(z_1,x) \\ m_2|h_1 &= K(h_1,z_2)K(z_2,z_2)^{-1}m_2^{\backslash 1} \\ v_2|h_1 &= \sigma_2 + K(h_1,h_1) - K(h_1,z_2)K(z_2,z_2)^{-1}K(z_2,h_1) + K(h_1,z_2)K(z_2,z_2)^{-1}V_2^{\backslash 1}K(z_2,z_2)^{-1}K(z_2,h_1) \end{aligned} \tag{3.10}$$

The 'law of integrated conditionals' can be used to moment match the integral above (Barber and Schottky, 1998; Deisenroth and Mohamed, 2012; Girard et al., 2003).

$$\begin{aligned} \mathcal{Z} &\approx \mathcal{N}(m_2, v_2) \\ m_2 &= E_{q(h_1)}[m_2|h_1] \\ v_2 &= E_{q(h_1)}[v_2|h_1] + var_{q(h_1)}(m_2|h_1) \end{aligned} \tag{3.11}$$

Expanded:

$$\begin{aligned} m_2 &= E_{q(h_1)}[K(h_1,z_1)]A \\ v_2 &= \sigma_1 + E_{q(h_1)}[K(h_1,h_1)] + tr(BE_{q(h_1)}[K(z_2,h_1)K(h_1,z_2)] - m_2^2 \\ A &= K(z_2,z_2)^{-1}m_2^{\backslash 1} \\ B &= K(z_2,z_2)^{-1}\left(V_2^{\backslash 1} + m_2^{\backslash 1}m_2^{\backslash 1,T}\right)K(z_2,z_2)^{-1} - K(z_2,z_2)^{-1} \end{aligned} \tag{3.12}$$

The expectation of the kernel matrix is not always analytically computable, but for squared exponential kernels, which we used in this project exclusively, it is (Titsias, 2009; Wilson and Adams, 2013).

All the equations can be naturally extended to more than two layers, although we only used two layers in our experiments.

In practice, the gradients of the ADF can be backpropagated through the layers using the chain rule, hence the title ‘backpropagation’, similarly to the backpropagation algorithm used in training neural networks (Hernández-Lobato and Adams, 2015). Fortunately, with the availability of automatic differentiation tools, such as Tensorflow, the analytical forms of the gradients can be computed automatically using repeated application of the chain rule. Section 3.3 gives more details on the development framework.

Moreover, the process is suitable for minibatch training to speed up convergence. Simply, in the final term, project the result of the minibatch to the whole.

$$log(p(y|\boldsymbol{u})) \approx \mathcal{F}(\alpha) = -(N-1)\phi(\theta) + N(\phi(\theta^{\setminus 1}) - \phi(\theta_{prior}) + \frac{N}{|B|}\sum_{b=1}^{|B|} log\mathcal{Z}_b \quad (3.13)$$

3.2.3 Joint DGPs

Our novel extension with Joint DGPs is to add a second output node to the network. This enables the DGP to have multiple outputs and therefore predict multiple objective functions simultaneously. The goal is to capture the underlying correlations in the objective functions and to reduce the number of models that need to be trained to one.

The addition of multiple output units is fairly straight-forward. We need to simply calculate the approximate marginal likelihoods and optimize their sum.

There is no change in computing the log normalizers Φ. As for $\mathcal{Z}_n$, the means m_2 and variances v_2 have to be calculated separately for the two output units and the resulting $log\mathcal{Z}_n$-s need to be summed up.

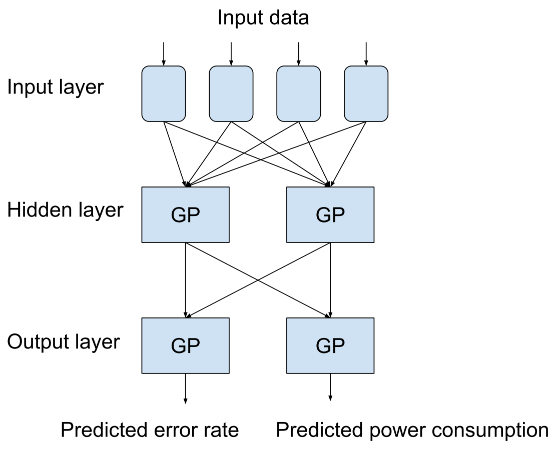


Fig. 3.3 The architecture of Joint DGPs.

3.3 Implementation details

The implementation consisted of four distinct parts: the implementation of the DGP and GP models, the implementation of the multiobjective optimization process and the support for distributed experiments.

3.3.1 Implementation of the DGP models

This section gives a brief overview on Automatic Differentiation Tools (ADT) followed by justification on why it was necessary to implement the DGP models in Tensorflow.

Automatic Differentiation Tools

The use of machine learning in industry and academia was revolutionized by the introduction of ADTs. ADTs are toolkits or libraries for machine learning that enable simple and very efficient implementations without requiring a significant technical background.

The key idea is that ADTs separate the light computation, such as processing the data, and the heavy computation, such as calculating matrices and derivatives. Following Amdahl's law, they focus on optimizing the performance of the latter.

They work by using a high-level programming language (Python in our case) to define the computation that needs to be carried out for a given model. Examples include calculating kernel matrices and outputs of the layers in a DNN. Then, they build a 'computation graph' that implements the computations in a low-level programming language (Tensorflow uses C++). Since the computation is defined first, the low-level code that executes it can be highly optimized for the hardware. In the final stage, when the training data is supplied, the computation can be executed very efficiently. Moreover, the setup allows for quick adaptation to different computing hardware as well as to use of GPUs.

A key feature of ADTs is that they can calculate the derivative of any expression as long as it was derived using differentiable steps. This is achieved by repeated application of the chain-rule. Since it is all automatic, the code for the computation can be more clear, concise and less error-prone.

Nowadays, the use of ADTs is widespread. They are essential tools for producing clear and easily modifiable code.

Tensorflow implementation

When we started the project, we had a working implementation of DGPs. This was the original code used for (Bui et al., 2016).

The original code was written in Python and used the Theano framework for computation. The shortcomings of this code were the lack of comments, the presence clutter and inefficiencies. The code was unclear and difficult to extend or modify. This prevented further use and adaptation.

An issue with Theano was that it required compilation when run on a machine for the first time. This could cause issues when running the code in a distributed fashion.

As the result of the shortcomings, we decided to write a new implementation that uses the Tensorflow framework. We chose Tensorflow because of its growing presence in the machine learning community and its ease of use. It not only allowed for clear and extensible code but also eliminated the need for compilation.

In Section 4.1, we demonstrated that new implementation in Tensorflow is more efficient than the original Theano one. It resulted in a $1.5-4x$ speedup depending on the number of training points. A large portion of the gains come from Tensorflow's more efficient initialization of the computation graph.

3.3.2 Implementation of the GP baseline models

Regarding the GP model that we used as the baseline in our experiments, we adopted a library implementation. It was not necessary to implement our own version due to the wide availability of GP implementations online. In addition, a library implementation provided a form of guarantee of correctness because it had been exposed to extensive testing therefore it was less likely to contain bugs.

We chose the GPy library[2] because its ease of use. Our positive prior experience with the library during the Kaggle contests also played a significant role in making the decision.

In terms of training the hyperparameters, the hyperparameter space can have many local minima therefore we optimized them using an l-BFGS (Liu and Nocedal, 1989) optimizer with 20 restarts. l-BFGS is a limited memory, gradient based optimizer that uses the Hessian matrix for quick convergence. The choice of the optimizer had little impact on the result.

3.3.3 Implementation of the Multiobjective Optimization process

The multiobjective optimization required implementation of the optimization framework and the data collection strategy.

Both of these were fairly straight-forward to implement in Python. The optimization process required iterations of model updates based on the acquisition values of the candidate points.

[2]https://sheffieldml.github.io/GPy/

The acquisition value was based on SMSego. SMSego took the prediction means and variances and calculated the hypervolume of the potential point.

The algorithm we used for calculating the hypervolume was based on recursion. There has been substantial research on efficient ways of calculating the hypervolume (While et al., 2006), but that is far beyond the scale of this project. We could afford to use an inefficient implementation due to the low number of dimensions and datapoints.

3.3.4 Support for distributed computing

Retraining the DGP model in every iteration is a costly computation. In order to finish the optimization experiments, it was necessary to be able to run multiple instance on a cluster.

We used the department's grid engine to run the experiments because it allowed for larger scale experiments than our Microsoft Azure credit did.

Due to issues with the grid, we had to build a robust setup for the experiments. Our processes frequently crashed due to power outages and insufficient storage space. To combat the problem we implemented automatic recovery of crashed experiments.

Chapter 4

Results

4.1 Verification of the implementation

The first set of experiments verified the implementation and tuned the training parameters. Our goal was to establish that the model is a working implementation of a DGP and to fix the initial learning rate, batch size and the number of iterations that lead to convergence.

Figure 4.1 shows the decreasing EP energy over the training iterations. The subtle fluctuations were due to the small batch size and learning rate, but we were still able to verify that the model was converging.

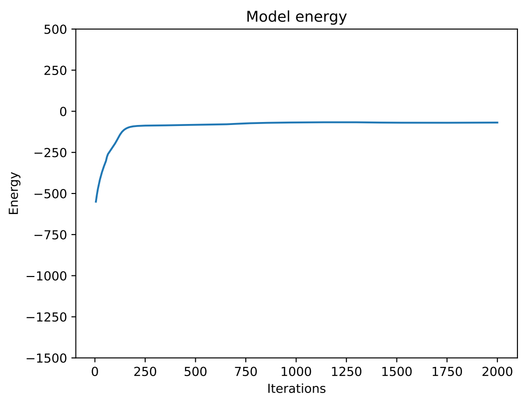
The training and test likelihoods are plotted on Figure 4.2. Both the training and test likelihoods converged rapidly, often reaching their maximum after as few as 500 epoch.

For some models, such as neural networks, overfitting can be a serious issue and there is a wealth of research published on combating the problem. When looking for signs of overfitting, we found that the test likelihood steadily increased through the training process which indicated that the model did not overfit. As a result, we were able to set the number of training epoch to a comfortably high number.

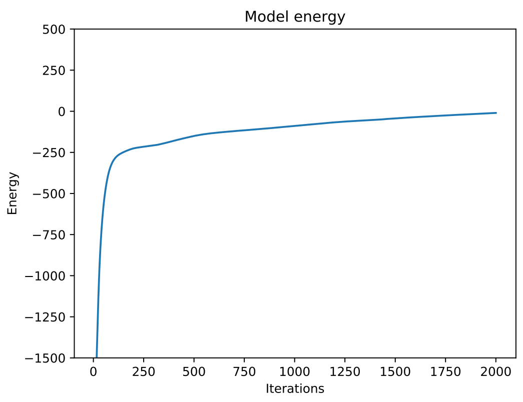
Our choice of stochastic optimizer was Adam (Kingma and Ba, 2014). We tested other gradient-based optimizers as well, but they all performed within close proximity of each other.

We settled on the training parameters shown in Table 4.1. They were proven to be adequate for the quantity and quality of the training data used during the optimization process.

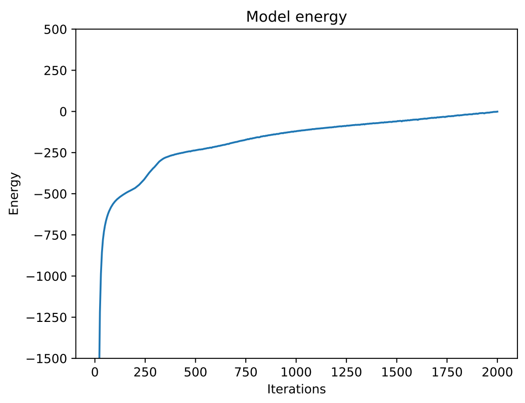
The selection of the training and test sets is described in Section 4.5.



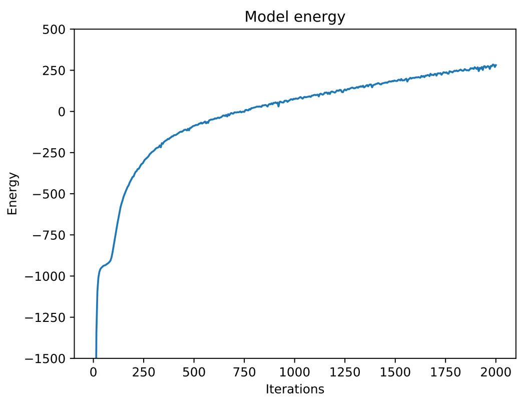
(a) $N = 20$.



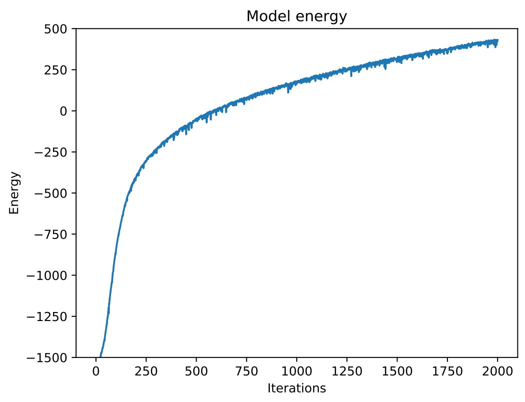
(b) $N = 50$.



(c) $N = 100$.

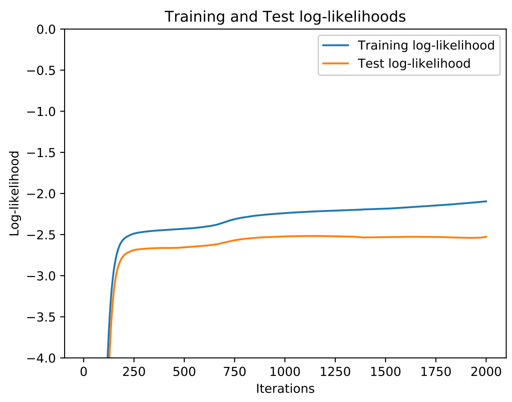


(d) $N = 200$.

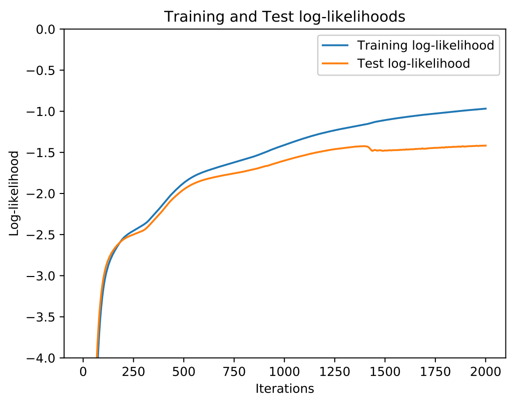


(e) $N = 300$.

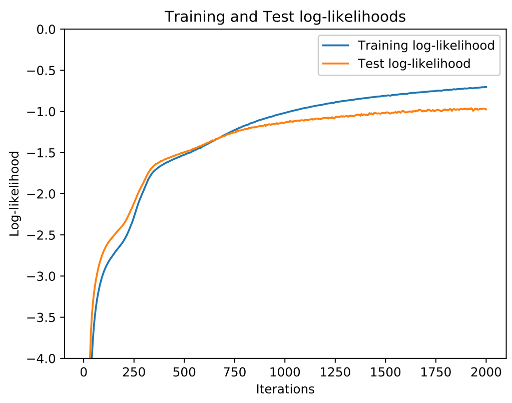
Fig. 4.1 Increasing EP energy during training.



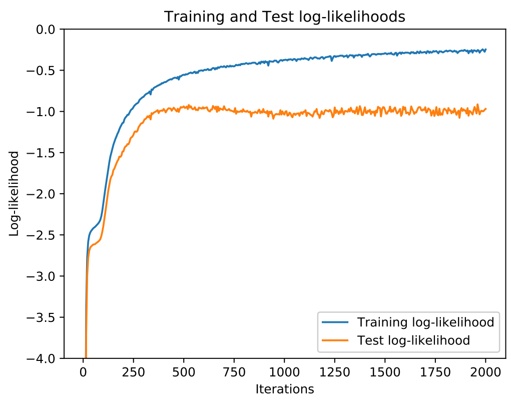
(a) $N = 20$.



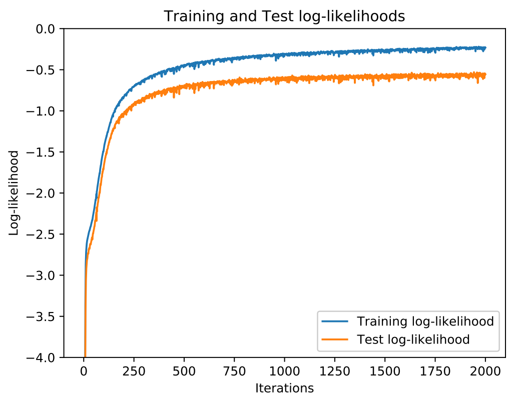
(b) $N = 50$.



(c) $N = 100$.



(d) $N = 200$.



(e) $N = 300$.

Fig. 4.2 Training and test log-likelihoods during training.

4.2 Runtime comparison

One of the reasons why we decided to re-implement DGPs from scratch in Tensorflow was the speedup that we were hoping to get from the state-of-the-art machine learning toolkit. In this experiment, we measured the benefits of the more optimized toolkit.

Initial learning rate	No. of training epoch	Batch size
0.01	2000	50

Table 4.1 Training parameters

The experiments were run on a 16 core Microsoft Azure machine with no other processes running during the tests. Multithreading was enabled and GPUs were not used. We checked that both processes could fit in RAM in order to ensure that the bottleneck was the processing power of the machine. The runtimes were very consistent therefore we deemed unnecessary to repeat the experiment multiple times.

Figure 4.3 shows the training times using the training parameters from Section 4.1. The Tensorflow implementation achieved a $1.5-4x$ speedup depending on the number of training samples. The speedup was more significant for fewer samples due to Tensorflow being able to initialize the computation graph significantly faster which took constant time regardless of the number of training samples.

The selection of the training and test sets is described in Section 4.5.

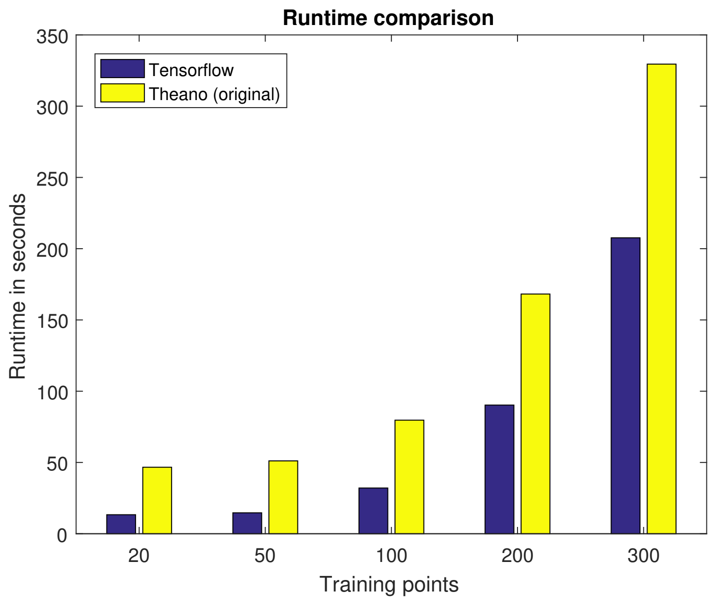


Fig. 4.3 Runtime of the training process for the two implementations.

4.3 Multiobjective Optimization

Our main experiments measured the improvement in hypervolume over the iterations of the multiobjective optimization process. This experiment draws the comparison between the predictive models in the setting of neural network hardware accelerator optimization.

The experiments were run for 200 iterations starting from an initial set of 30 randomly selected evaluations. For each model, we ran 50 experiments in parallel. Moreover, to reduce the variance in the results, the initial set of 30 datapoints were shared across the models but each of the 50 experiments had a different initial set.

Along with the means, the t-based confidence intervals for the mean [1] hypervolume are plotted in Figure 4.4 for each model. Table 4.2 contains the same information in a numeric format.

We selected 6 models that we wanted to compare. Three baselines: Random, Squared exponential GP, and Matérn GP as well as three novice models: Shallow GP, DGP, and Joint DGP.

Random

In the random model, the point of the next evaluation was selected randomly from all candidate points. The process was expected to improve the hypervolume since there was always the possibility of selecting Pareto-optimal points.

The random process needed to be included in the baseline models since the hypervolume was expected to increase even by random evaluations. To clarify, we expected all models to exceed the performance of a random process, we included it in order to demonstrate the effectiveness of the intelligent decision making in the multiobjective optimizations process.

Squared exponential GP

The Gaussian Process (described in Section 2.2) using a squared exponential function was the vanilla GP model. It was straight-forward to implement and it worked reasonably well in all settings.

The kernel was a squared exponential function with a separate length-scale for each dimension. The hyperparameters were optimized using ML training.

[1] http://www.stat.wmich.edu/s216/book/node79.html

Matérn GP

Our second GP model used a Matérn kernel function with $\nu = \frac{5}{2}$. This was the model that Spearmint, the toolkit for multiobjective optimization, used.

The Matérn covariance function is often better suited for optimization than a simple SE because it does not drop off as quickly for distant points. This allows points that are further away to influence the model prediction, which is desirable when the data is sparse.

The hyperparameters were trained using ML.

Sparse GP

Sparse GP is a GP that employs the FITC approximation to set the number of inducing points to 10% of the training points.

This model had a squared exponential kernel function and was trained using MAP training.

The FITC sparse approximation is often used to reduce the cost of GPs. We included this model in the experiments to examine the impact of the model degradation caused by the sparsity approximation.

DGP

This is the Deep Gaussian Process model described in Section 3.2.1.

It uses a separate DGP model for each objective function. They contain a single hidden layer with two hidden nodes and a single output node. The hyperparameters were trained using EP.

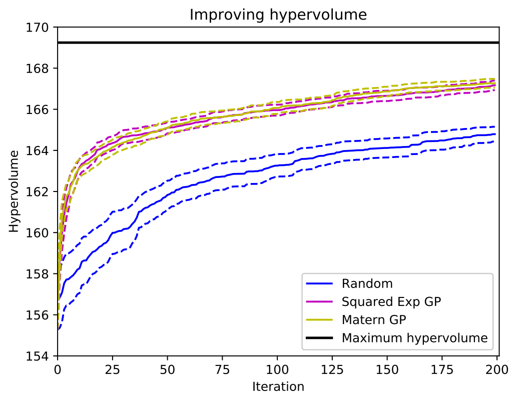
Every node had squared exponential kernel function and the number of inducing points was set to 10% of the training points.

Joint DGP

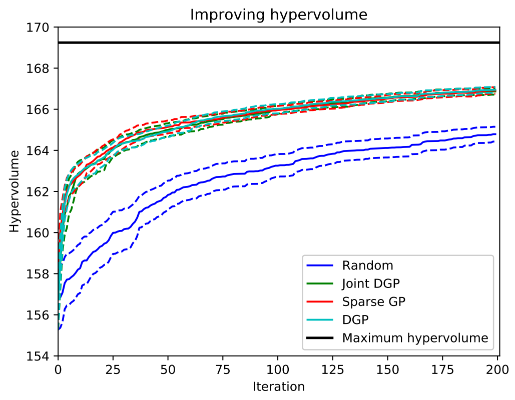
Our extension to DGPs was the use multiple output units in the same model. That gave Joint DGPs the ability to predict all objective functions in a single model.

The model had a single hidden layer with two hidden nodes and two output nodes in the output layer. Each output node predicted a different objective function (predictive error and power consumption).

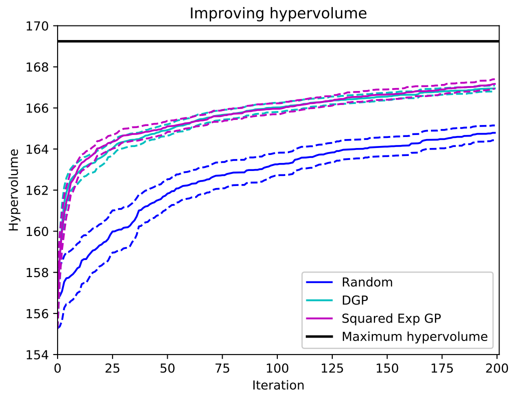
Every node had squared exponential kernel function and the number of inducing points was set to 10% of the training points.



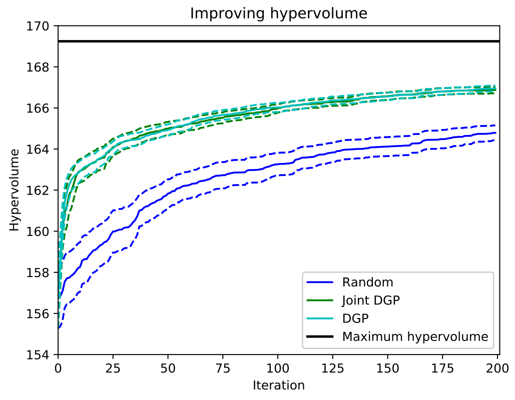
(a) Comparison of Squared Exp GP and Matérn GP.



(b) Comparison of Sparse GP, DGP and Joint DGP models.



(c) Comparison of DGP and Squared Exp GP.



(d) Comparison of DGP and Joint DGP.

Fig. 4.4 The increasing hypervolume over the iterations.

Iterations	20	50	100	200
Random	159.35 ± 1.02	161.83 ± 0.70	163.26 ± 0.55	164.80 ± 0.36
Squared Exp GP	164.10 ± 0.35	165.10 ± 0.27	165.96 ± 0.27	167.16 ± 0.23
Matérn GP	163.85 ± 0.38	165.09 ± 0.32	166.07 ± 0.29	167.28 ± 0.21
Sparse GP	163.81 ± 0.39	165.14 ± 0.31	165.96 ± 0.20	166.96 ± 0.15
DGP	163.60 ± 0.40	164.93 ± 0.28	166.03 ± 0.22	166.88 ± 0.14
Joint DGP	163.57 ± 0.54	165.00 ± 0.32	165.98 ± 0.21	166.85 ± 0.14

Table 4.2 The increasing hypervolume over the iterations.

4.4 Pair-wise comparison of the hypervolume improvement

For a more conclusive analysis of the hypervolume improvement, we did a pairwise comparison between the models.

The justification for this type of analysis is that the variance in the results of the previous experiment originated from two sources. Firstly, the randomness in the models. Random initialization and differences in the stochastic optimization can lead to slightly different models in each training. Secondly, the variance came from the different set of initial 30 points for each of the 50 runs. In order to eliminate variance from the second source, we can analyze the pairwise performance of the models.

One can derive p-values for one model's superiority over another by examining how many times, out of the 50 runs, one model performed better than the other. The p-values are calculated under the null hypothesis that the two models performed equivalently well and they had equal chances of overtaking the other. From this assumption, the p-values can be calculated using the biased-coin test. If, out of 50 runs, one model outperformed another in X occasions then the associated p-value can be obtained using:

$$p = \sum_{k=0}^{X} \binom{50}{k} 0.5^{50}$$

Since we are making many comparisons at the same time, we consider statistical significance at $0.01 > p$ or $0.99 < p$ which is equivalent to a ratio under $\frac{16}{50}$. This is only met by the comparison between the random process and the squared exponential GP.

A final note is that in the case of 200 iterations, the experiments do not add up to 50. Despite our best efforts, two experiments crashed before reaching 200 iterations. More details on this in Section 3.3. The p values were adjusted under the assumption that the crashes were independent from the initial sets.

		20 iterations			50 iterations			100 iterations			200 iterations		
Model 1	Model 2	M1	M2	p	M1	M2	p	M1	M2	p	M1	M2	p
Random	Squared Exp GP	3	47	**0.000**	3	47	**0.000**	2	48	**0.000**	1	47	**0.000**
Squared Exp GP	DGP	31	19	0.968	25	25	0.556	20	30	0.101	25	23	0.667
Squared Exp GP	Matérn GP	27	23	0.760	20	30	0.101	22	28	0.240	23	25	0.443
Sparse GP	Squared Exp GP	18	32	0.032	27	23	0.760	24	26	0.444	20	28	0.156
DGP	Joint DGP	25	25	0.556	19	31	0.059	24	26	0.444	22	26	0.333

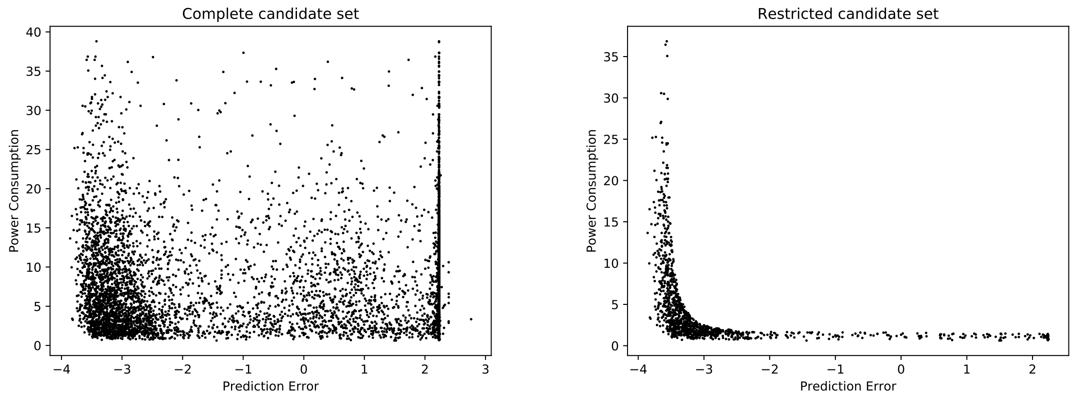
Table 4.3 Pairwise comparison of the models.

4.5 Model accuracies

To further our investigations, we compared the predictive power of our models. We examined two metrics: log-likelihood and Root Mean Squared Error (RMSE). The goal of this experiment was to assess how well the models were able to fit the training data independently of the optimization process.

$$RMSE = \sqrt{\sum_{n=1}^{N} \frac{(\tilde{y_n} - y_n)^2}{N}}$$

Since during the optimization process, ‘close-to-optimal’ points are evaluated more often and the model accuracies are more important for these points, we tested the models on a restricted set of points. We ordered all candidate points by the number of points that they were dominated by. Then, we selected the first 20% of the points as the restricted set. Figure 4.5 demonstrates the concept. When running the experiment, we randomly sampled the training points and used the remainder of the restricted set as test points.



(a) The complete candidate set. (b) The restricted candidate set.

Fig. 4.5 The ‘close-to-optimal’ points.

The test log-likelihoods and the test RMSE are reported in Table 4.4 and 4.5 respectively. The 95% confidence intervals were calculated using the Student-t distribution for the means. The scores are reported separately for the two objectives as well as an overall score, which is the average of the two.

Training Points	Squared Exp GP	Matérn GP	Sparse GP	DGP	Joint DGP
Predictive Error					
20	-20.94 ± 4.02	-16.26 ± 3.95	-1.96 ± 0.13	**-2.11 ± 0.23**	-2.32 ± 0.28
50	-6.77 ± 1.49	-4.26 ± 0.73	-1.51 ± 0.05	**-1.44 ± 0.08**	-1.69 ± 0.17
100	-2.87 ± 0.33	-2.42 ± 0.33	-1.16 ± 0.03	-1.04 ± 0.10	**-0.95 ± 0.07**
200	-1.81 ± 0.12	-1.50 ± 0.12	-0.93 ± 0.03	**-0.82 ± 0.07**	-0.84 ± 0.09
300	-1.56 ± 0.10	-1.25 ± 0.08	-0.82 ± 0.03	**-0.56 ± 0.08**	-0.58 ± 0.09
Power Consumption					
20	-15.53 ± 2.47	-10.83 ± 2.19	**-2.27 ± 0.11**	-2.67 ± 0.38	-2.30 ± 0.18
50	-5.26 ± 0.68	-3.72 ± 0.75	-1.81 ± 0.04	-1.64 ± 0.07	**-1.55 ± 0.06**
100	-3.55 ± 0.64	-2.42 ± 0.53	-1.29 ± 0.03	-1.14 ± 0.03	**-1.06 ± 0.03**
200	-2.16 ± 0.16	-1.45 ± 0.08	-0.95 ± 0.02	-0.84 ± 0.02	**-0.76 ± 0.02**
300	-1.69 ± 0.10	-1.14 ± 0.09	-0.73 ± 0.02	-0.66 ± 0.03	**-0.37 ± 0.04**
Overall					
20	-18.24 ± 2.26	-13.55 ± 2.18	-2.11 ± 0.08	-2.39 ± 0.22	**-2.31 ± 0.16**
50	-6.01 ± 0.77	-3.99 ± 0.50	-1.66 ± 0.03	**-1.54 ± 0.05**	-1.62 ± 0.09
100	-3.21 ± 0.38	-2.42 ± 0.33	-1.22 ± 0.02	-1.09 ± 0.05	**-1.01 ± 0.04**
200	-1.99 ± 0.11	-1.48 ± 0.08	-0.94 ± 0.02	-0.83 ± 0.04	**-0.80 ± 0.05**
300	-1.62 ± 0.07	-1.20 ± 0.06	-0.78 ± 0.02	-0.61 ± 0.04	**-0.48 ± 0.05**

Table 4.4 Log-likelihoods of the different predictive models.

Training Points	Squared Exp GP	Matérn GP	Sparse GP	DGP	Joint DGP
Predictive Error					
20	1.56 ± 0.06	**1.52 ± 0.06**	1.53 ± 0.05	1.61 ± 0.03	1.58 ± 0.02
50	1.17 ± 0.04	**1.14 ± 0.04**	1.25 ± 0.02	1.28 ± 0.04	1.33 ± 0.03
100	1.00 ± 0.02	0.96 ± 0.03	1.03 ± 0.02	**0.93 ± 0.03**	1.08 ± 0.04
200	0.87 ± 0.01	0.83 ± 0.02	0.89 ± 0.01	**0.75 ± 0.02**	0.83 ± 0.02
300	0.81 ± 0.01	0.75 ± 0.01	0.83 ± 0.01	**0.66 ± 0.01**	0.73 ± 0.02
Power Consumption					
20	3.18 ± 0.20	**2.91 ± 0.18**	3.25 ± 0.11	3.72 ± 0.13	3.92 ± 0.18
50	2.17 ± 0.14	**1.80 ± 0.11**	2.61 ± 0.05	2.91 ± 0.07	2.98 ± 0.08
100	1.80 ± 0.09	**1.23 ± 0.06**	2.19 ± 0.04	2.11 ± 0.07	2.21 ± 0.05
200	1.33 ± 0.06	**0.88 ± 0.03**	1.79 ± 0.04	1.49 ± 0.05	1.60 ± 0.04
300	1.20 ± 0.06	**0.83 ± 0.04**	1.48 ± 0.04	1.11 ± 0.04	1.22 ± 0.05
Overall					
20	2.53 ± 0.13	**2.34 ± 0.11**	2.55 ± 0.07	2.87 ± 0.08	3.00 ± 0.12
50	1.77 ± 0.09	**1.53 ± 0.07**	2.05 ± 0.04	2.25 ± 0.05	2.31 ± 0.05
100	1.47 ± 0.06	**1.12 ± 0.03**	1.71 ± 0.03	1.64 ± 0.04	1.75 ± 0.03
200	1.13 ± 0.04	**0.86 ± 0.02**	1.41 ± 0.02	1.18 ± 0.03	1.28 ± 0.03
300	1.03 ± 0.04	**0.80 ± 0.02**	1.20 ± 0.02	0.92 ± 0.03	1.01 ± 0.03

Table 4.5 RMSE of the different models

Chapter 5

Discussion

This chapter interprets the results of the experiments and discusses some of the issues and difficulties with the testing methodology.

5.1 Multiobjective Optimization

The multiobjective optimization experiment demonstrated the power of intelligent decision making. All models decisively outperformed the random process which shows that it is an effective solution to the hardware design problem.

The hypervolume exhibited a rapid increase in the first ~15 iterations which turned into a steady increase between iteration 20 and 130. The hypervolume nearly leveled out before iteration 150 and all methods reached a proximity of the optimal solution.

The experiment did not manage to distinguish between the six models. All models performed within the error bounds of the baseline GP model. This indicates that the models are expected to perform similarly well when put to the test in a real-world optimization process. However, there are a few key points that need to be mentioned relating to these results.

Potential causes of the similar results

This section examines the factors that could have played a role in the near identical hypervolume curves for the 6 models.

The first point to note is that the resolution, i.e. frequency, of the set of candidate points can cause issues. It is important to have many candidate points near the Pareto-frontier in order for the methods to make an actual choice on the next evaluation. If the resolution is too low then the methods cannot distinguish themselves by making a slightly better decisions since a slightly better model might just arrive to the same candidate point. This likely did not affect our experiment since we ran for very few iterations (200 iterations compared to 6276 candidate points), however, this can become an issue for higher iterations.

The second source of variance is the noise in the data itself. Since the data was obtained by running simulations, it inevitably contains measurement noise. This means that an inferior method can prevail just by being more fortunate with its evaluations. By random chance, it could stumble upon points that scored very well in the objectives due to the measurement noise. The model accuracy experiment (Section 4.5) did confirm that the prediction error was very difficult to predict which we think played a significant role in the outcome.

Thirdly, the initial set of evaluations can cause differences in the methods. It is a definite advantage to have near Pareto-optimal points in the initial set. To combat the problem, we used the same sets of initial points across the six models but a different one for each of the 50 runs for a single model. This did reduce the variance introduced by the initial set but it did not eliminate it completely. Furthermore, the hypervolume error bars are inadequate tools for assessing performance when the starting points are shared across the runs. A more appropriate assessment is presented in the form of a pair-wise comparison of models in Section 4.4. This experiment completely eliminated the variance introduced by the initial since it only considered the binary state of having higher hypervolume between models that shared the initial set. The experiment further supported our previous observations which is that all models outperform the random process but there is no significant difference between the models themselves.

Reference point

During experimentation, we found that the effectiveness of the process was strongly influenced by the choice of the reference point. The reference point determined the way we calculated the hypervolume during the process (Section 2.1.3). Initially, as (Emmerich and Naujoks, 2004) suggested, we fixed the reference point at $(max_{error} + 1.0, max_{power} + 1.0)$. This was an inadequate choice and it caused all models to perform no better than the random process. After the investigation we identified the problem. The prediction error and the power consumption values lied on different scales and the hypervolume did not account for that. The choice of reference point overwhelmingly prioritized low error over power consumption since power consumption worked on a larger scale. The issue was further amplified by the fact that that the error rate is the more difficult metric to predict in general (Section 4.5).

We compiled the considerations when choosing the reference point into Table 5.1. Our suggestion is not to rely on a rule of thumb such as $(max_{error} + 1.0, max_{power} + 1.0)$. The reference point needs to be hand-picked for the problem because it plays a key role in deciding the nature of the Pareto-frontier that we get as the result. For our experiments, we used $(10.0, 15.0)$ (Figure 1.1) because it lies in the 'sweet-spot' that does not prioritize one objective function over the other.

	High in Error	Low in Error
High in Power	Preference for points with extremely low values in error or power. Balanced points are not prioritized.	Preference for low power. Error values are not prioritized.
Low in Power	Preference for low error. Power values are not prioritized.	Preference for points with low power and error. Points with extremes in one metric are not prioritized.

Table 5.1 Considerations when choosing the reference point.

Scale considerations

As we mentioned it in Section 4.3, we ran the experiment 50 times for each model for 200 iterations. Considering that the computational cost all of our models grew cubically, the cost of running the experiment was substantial.

We would have preferred to run the experiment in larger scale and for more iterations, however, after examining the results, we decided against doing so. The experiments conclusively indicate that there are no significant differences in the models and running them in larger scale would change this fact. As for running the experiments for more iterations, the graphs indicate that the problem is effectively 'solved' after ~100 iterations and the hypervolume levels out. Moreover, having too many iteration amplifies the resolution problem mentioned earlier.

To get an insight into the performance of the models with more training points, we can refer to the model accuracy experiments is Section 4.5. They do not suggest that we can expect any change if there were more iterations.

We concluded that the cost of running the experiment in a larger scale outweighed the potential benefits.

Experiment conclusions

From this experiment, we made the conclusion that all models performed similarly in the multiobjective optimization setting. We identified two potential causes: the resolution of the candidate set and the high noise in the data.

Using pairwise comparisons, we showed that the third potential cause, the differences in initialization, did not play a significant role.

5.2 Model accuracies

Our second set of experiments focused on solely the comparison of the models. The training and test sets were restricted to the near Pareto-optimal points and we measured two metrics: log-likelihood and RMSE (Section 4.5).

The reason that this experiment is needed is that there are multiple factors that play a role in the even outcome of multiobjective optimization experiment. Here, we purposefully eliminated the surrounding optimization process. While this did allow us to distinguish between the accuracy of the predictions made by the models, the outcome of this experiment did not justify making conclusions about the optimization process as a whole.

The log-likelihoods and RMSE values are shown in Table 4.4 and 4.5 respectively.

Log-likelihoods

Looking at the log-likelihoods, the best models were the DGP, Joint DGP followed by the Sparse GP. These three were better than the Squared Exp GP and Matérn GP baselines by a significant margin for all objectives and number of training samples.

DGP and Joint DGP outperformed Sparse GP for ≥ 100 training points in a statistically significant way. This demonstrates the flexibility of the deep model.

DGP and Joint DGP were head-to-head up until 300 training points. At 300 points, the Joint DGP model surpassed the DGP model by having an impressive -0.37 log-likelihood for power consumption.

RMSE

The RMSE experiment gives us insight into the quality of the prediction means since RMSE does not consider the uncertainty.

The results are different from the log-likelihood experiment. The Matérn GP model performed the best overall and Sparse GP the worst with the rest in-between the two.

Scale considerations

Since this experiment required less models to be trained than the optimization experiment, we could run each test 100 times with randomized training samples. This led to tight confidence bounds and allowed us to differentiate the models.

Experiment conclusions

The combined result of the two experiments indicate that the deep models were on-par with the GP models when it came to predicting the mean, but they were better at predicting the uncertainty. This lead to similar RMSE values but higher log-likelihoods for the DGP models.

The best performing model in term of log likelihood were the DGP and the Joint DGP models. When it came to RMSE, the GP with a Matérn kernel proved to be superior to all other models.

5.3 Review of the models

This section discusses each model separately.

5.3.1 GP models

We used two GPs, one with a squared exponential kernel and one with a Matérn kernel, as baselines for our experiments. As discussed earlier, they performed reasonably well on the task.

The only difference we found between the two is that the RMSE was lower for the Matérn GP. This did not prove to be significant enough to provide an edge in the optimization process.

5.3.2 Sparse GP models

As one of our models, we included a Spare GP in order to examine the impact of the FITC approximation. We found that the Sparse GP model surpassed the GP model in log-likelihood, but it was the worst model when it came to RMSE.

This leads to the conclusion that the approximation had a negative impact on the prediction mean and a positive one on the variance. The explanation for the former is the fact that we are approximating the GP model with only a few inducing inputs. As for the latter, the sparse model was able to better model the uncertainty using the flexibility of the inducing points.

This model demonstrated the applicability of sparse approximations even at very low number of datapoints since the impact in the optimization process was not measurable.

5.3.3 DGP and Joint DGP models

DGPs and Joint DGPs matched the performance of the GP baselines in the multiobjective optimization experiment. This proves that they are a viable choice for the task at hand.

The introduction of hidden layers lead to an improvement in the models. The log-likelihoods experiment demonstrated that the deep models were able to give better uncertainty estimates than the shallow (single layer) counterparts. However, this was not the case in the RMSE experiment. The accuracy of the prediction means was comparable to GPs with slight differences depending on the number of training points.

The improved uncertainty estimates did not prove to be enough to lead to a significant improvement in the optimization task due to factors described in Section 5.1.

As for the Joint DGP models, they exhibited substantial improvement in the log-likelihood of the power consumption at 300 training points which lead to a small improvement in the overall log-likelihood. The effects on the log-likelihood of the predictive error were minimal. While this proves the idea that the joint predictor is able to capture the underlying correlations in the objective function, the improvement was, again, not enough to make a significant impact in the optimization task.

Chapter 6

Conclusions

In this project, we successfully implemented DGP models and applied them to the real-world problem of optimizing neural network hardware accelerators.

In the optimization task, the DGP model performed on par with the baseline GP models that are widely used both in academia and in the industry. Moreover, the DGP models managed to surpass GPs in terms of test log-likelihood on the data.

As a novel extension, we experimented with Joint DGP models that used a single predictor for estimating all the objective functions. We showed that Joint DGPs improved the uncertainty estimates over the standard DGP model in the presence of at least 100 training samples.

6.1 Further work

Further work based on the results of this project can be divided into three main categories.

6.1.1 Application to other problems and domains

Bayesian optimization is known for its wide range of applications and the same applies to multiobjective optimization.

The most straight-forward application domain is hardware design and robotics. Bayesian Optimization has a good historical record on these tasks. To mention a few examples, (Lizotte et al., 2007) and (Calandra et al., 2016) both discuss the problem of gait optimization in robotics. These works can be extended to use DGP models instead of single layer GPs.

Multiobjective optimization also has applications in machine learning. We used neural network hardware accelerators as our problem of focus, however, the same methodology is applicable to deep learning itself. The models often have a large hyperparameter space that is costly to evaluate and optimization can offer a solution to this. (Snoek et al., 2012) offers an in-depth investigation on the problem.

6.1.2 Improving the optimization process on the current task

This project gave an insight on the applicability of DGP models in an SMSego optimization process. There are several other approaches that can be tested.

One direction is to improve the data collection strategy. SMSego was a straight-forward choice, but it might be the case that other approaches outperform SMSego. For example, unlike SMSego, EHI (Emmerich and Klinkenberg, 2008) is able to exploit non-Gaussian predictions. This might give EHI an edge when the predictive model is able to output a non-Gaussian posterior. For example, Bayesian Neural Networks (Depeweg et al., 2016) are an example of such models.

The second direction is to test different predictive models. As mentioned earlier, Bayesian Neural Networks can be a good candidate since they are able to cope with sparse data and noisy measurements.

6.1.3 Improving the DGP model

DGPs can be improved by making them more efficient or more accurate.

For example, (Salimbeni and Deisenroth, 2017) improves on the flexibility of DGPs by allowing correlations between layers. This can increase the quality of the uncertainty estimates of DGPs.

Efficiency can be improved by further optimizing hyperparameters such as the dept and width of the network as well as the number of inducing points per node. In fact, this optimization task is an excellent candidate for Bayesian Optimization.

References

David Barber and Bernhard Schottky. Radial basis functions: a bayesian treatment. In *Advances in Neural Information Processing Systems*, pages 402–408, 1998.

Thang Bui, Daniel Hernández-Lobato, Jose Hernandez-Lobato, Yingzhen Li, and Richard Turner. Deep gaussian processes for regression using approximate expectation propagation. In *International Conference on Machine Learning*, pages 1472–1481, 2016.

Roberto Calandra, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. Bayesian optimization for learning gaits under uncertainty. *Annals of Mathematics and Artificial Intelligence*, 76(1-2):5–23, 2016.

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

Andreas Damianou and Neil Lawrence. Deep gaussian processes. In *Artificial Intelligence and Statistics*, pages 207–215, 2013.

Marc Deisenroth and Shakir Mohamed. Expectation propagation in gaussian process dynamical systems. In *Advances in Neural Information Processing Systems*, pages 2609–2617, 2012.

Stefan Depeweg, José Miguel Hernández-Lobato, Finale Doshi-Velez, and Steffen Udluft. Learning and policy search in stochastic dynamical systems with bayesian neural networks. *arXiv preprint arXiv:1605.07127*, 2016.

Michael Emmerich and Jan-willem Klinkenberg. The computation of the expected improvement in dominated hypervolume of pareto front approximations. *Rapport technique, Leiden University*, 34, 2008.

Michael Emmerich and Boris Naujoks. Metamodel-assisted multiobjective optimisation strategies and their application in airfoil design. *Adaptive computing in design and manufacture VI*, 6:248–260, 2004.

Agathe Girard, Carl Edward Rasmussen, Joaquin Quinonero Candela, and Roderick Murray-Smith. Gaussian process priors with uncertain inputs application to multiple-step ahead time series forecasting. In *Advances in neural information processing systems*, pages 545–552, 2003.

Rafael Gómez-Bombarelli, David Duvenaud, José Miguel Hernández-Lobato, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *arXiv preprint arXiv:1610.02415*, 2016.

Daniel Hernández-Lobato, Jose Hernandez-Lobato, Amar Shah, and Ryan Adams. Predictive entropy search for multi-objective bayesian optimization. In *International Conference on Machine Learning*, pages 1492–1501, 2016a.

José Miguel Hernández-Lobato and Ryan Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. In *International Conference on Machine Learning*, pages 1861–1869, 2015.

José Miguel Hernández-Lobato, Michael Gelbart, Matthew Hoffman, Ryan Adams, and Zoubin Ghahramani. Predictive entropy search for bayesian optimization with unknown constraints. In *International Conference on Machine Learning*, pages 1699–1707, 2015.

José Miguel Hernández-Lobato, Michael A Gelbart, Ryan P Adams, Matthew W Hoffman, and Zoubin Ghahramani. A general framework for constrained bayesian optimization using information-based search. 2016b.

José Miguel Hernández-Lobato, Michael A Gelbart, B Reagen, Robert Adolf, Daniel Hernández-Lobato, Paul N Whatmough, David Brooks, Gu-Yeon Wei, and Ryan P Adams. Designing neural network hardware accelerators with decoupled objective evaluations. In *NIPS workshop on Bayesian Optimization*, 2016c.

JM Jin and Zhang Shan Jjie. *Computation of special functions*. Wiley, 1996.

Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Joshua Knowles. Parego: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66, 2006.

Yingzhen Li, José Miguel Hernández-Lobato, and Richard E Turner. Stochastic expectation propagation. In *Advances in Neural Information Processing Systems*, pages 2323–2331, 2015.

Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

Daniel J Lizotte, Tao Wang, Michael H Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. In *IJCAI*, volume 7, pages 944–949, 2007.

Thomas Peter Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, 2001.

Benjamin Recht James Demmel Orianna DeMasi, Joseph Gonzalez. Using bayesian optimization for hardware design. 2014.

Victor Picheny. Multiobjective optimization using gaussian process emulators via stepwise uncertainty reduction. *Statistics and Computing*, 25(6):1265–1280, 2015.

Wolfgang Ponweiser, Tobias Wagner, Dirk Biermann, and Markus Vincze. Multiobjective optimization on a limited budget of evaluations using model-assisted\mathcal {S}-metric selection. In *International Conference on Parallel Problem Solving from Nature*, pages 784–794. Springer, 2008.

Joaquin Quiñonero-Candela and Carl Edward Rasmussen. A unifying view of sparse approximate gaussian process regression. *Journal of Machine Learning Research*, 6(Dec): 1939–1959, 2005.

Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 267–278. IEEE Press, 2016.

Hugh Salimbeni and Marc Deisenroth. Doubly stochastic variational inference for deep gaussian processes. *arXiv preprint arXiv:1705.08933*, 2017.

Matthias Seeger. Expectation propagation for exponential families. Technical report, 2005.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 97–108. IEEE, 2014.

Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264, 2006.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Michalis K Titsias. Variational learning of inducing variables in sparse gaussian processes. In *International Conference on Artificial Intelligence and Statistics*, pages 567–574, 2009.

Michalis K Titsias and Neil D Lawrence. Bayesian gaussian process latent variable model. In *International Conference on Artificial Intelligence and Statistics*, pages 844–851, 2010.

Lyndon While, Philip Hingston, Luigi Barone, and Simon Huband. A faster algorithm for calculating hypervolume. *IEEE transactions on evolutionary computation*, 10(1):29–38, 2006.

Christopher KI Williams and Carl Edward Rasmussen. Gaussian processes for regression. In *Advances in neural information processing systems*, pages 514–520, 1996.

Andrew Wilson and Ryan Adams. Gaussian process kernels for pattern discovery and extrapolation. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1067–1075, 2013.

Marcela Zuluaga, Guillaume Sergent, Andreas Krause, and Markus Püschel. Active learning for multi-objective optimization. In *International Conference on Machine Learning*, pages 462–470, 2013.

Appendix A

Derivation of the prediction mean and variance

We have

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y}' \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\boldsymbol{x},\boldsymbol{x}) & K(\boldsymbol{x},\boldsymbol{x}') \\ K(\boldsymbol{x}',\boldsymbol{x}) & K(\boldsymbol{x}',\boldsymbol{x}') \end{bmatrix}\right)$$

An alternative notation:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right)$$

The since the distribution is multivariate Gaussian, the conditional distribution of $x_2|x_1$ must also be a multivariate Gaussian. Therefore the objective is to derive the mean and the covariance matrix of the conditional distribution.

Consider $\boldsymbol{z} = x_2 + Ax_1$ where $A = -\Sigma_{21}\Sigma_{11}^{-1}$

$$\begin{aligned} cov(z,x_1) &= cov(x_2 + Ax_1, x_1) \\ &= cov(x_2,x_1) + Acov(x_1,x_1) \\ &= \Sigma_{21} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{11} \\ &= 0 \end{aligned} \tag{A.1}$$

z and x_1 are jointly Gaussian and they are uncorrelated therefore they are independent.

$$\begin{aligned} E(x_2|x_1) &= E(z - Ax_1|x_1) \\ &= E(z) - Ax_1 \\ &= \mu_1 + A\mu_2 - Ax_1 \\ &= \mu_1 - \Sigma_{21}\Sigma_{11}^{-1}(\mu_2 - x_1) \end{aligned} \tag{A.2}$$

As for the variance, the term Ax_1 is fully defined by x_1 therefore adding it to a conditional variance term will not affect it:

$$var(x_2|x_1) = var(x_2 + Ax_1|x_1) = var(z|x_1) = var(z)$$

Using the sum rule for the variance:

$$\begin{aligned} var(z) &= var(x_2 + Ax_1) \\ &= var(x_2) + Avar(x_1)A^T + cov(x_2, x_1)A^T + Acov(x_1, x_2) \\ &= \Sigma_{22} + \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{11}\Sigma_{11}^{T-1}\Sigma_{21}^T - \Sigma_{21}\Sigma_{11}^{T-1}\Sigma_{21}^T - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12} \\ &= \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12} \end{aligned} \tag{A.3}$$

Therefore we have,

$$x_2|x_1 \sim \mathcal{N}\left(\mu_1 - \Sigma_{21}\Sigma_{11}^{-1}(\mu_2 - x_1), \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}\right)$$

Appendix B

Expectation Propagation

Expectation Propagation (EP) Minka (2001) is an approximate method for calculating the posterior as well as approximating the marginal likelihood. It works from the assumption that the posterior distribution $q(\boldsymbol{u})$ takes the form:

$$q(\boldsymbol{u}) \propto p(\boldsymbol{u}) \prod_n \tilde{t}_n(\boldsymbol{u})$$

.

Where $p(\boldsymbol{u})$ is the prior and $\tilde{t}_n \quad \text{for} \quad n = 1, \ldots, N$ are approximate data factors. The idea is that each $\tilde{t}_n$ captures the contribution that a single datapoint makes to the posterior. The goal is to replace the intractable posterior with a product where each term is from a tractable distribution. In this project, $\tilde{t}_n$ always takes the form of an unnormalized multivariate Gaussian.

It would be ideal to optimize $KL(p(\boldsymbol{u}|x,y)||q(\boldsymbol{u}))$, however, that is generally not tractable. Instead, EP optimizes the data factors $\tilde{t}_n$ in an iterative process (Algorithm 2).

```
Data: $(x_n, y_n)$
$\tilde{t}_n \leftarrow$ random initialization
$q(\boldsymbol{u}) \leftarrow p(\boldsymbol{u}) \prod_n \tilde{t}_n(\boldsymbol{u})$
repeat
    for $n \leftarrow 1$ to $N$ do
        $q^{\setminus n}(\boldsymbol{u}) \propto \frac{q(\boldsymbol{u})}{\tilde{t}_n(\boldsymbol{u})} = p(\boldsymbol{u}) \prod_{i \neq j} \tilde{t}_n(\boldsymbol{u})$
        $\tilde{t}_n \leftarrow argmin_{t_n} KL\left(q^{\setminus n}(\boldsymbol{u}) p(y_n|x_n, \boldsymbol{u}) || q^{\setminus n}(\boldsymbol{u}) \tilde{t}_n(\boldsymbol{u})\right)$
        $q(\boldsymbol{u}) \leftarrow \tilde{t}_n(\boldsymbol{u}) q^{\setminus n}(\boldsymbol{u})$
    end
until convergence or divergence;
```

Algorithm 2: Iterative EP

This works because the KL divergence can be optimized analytically assuming multivariate Gaussian data-factors. An issue, however, is that the process is not guaranteed to converge. It is possible that the loop repeats infinitely through a cycle of data-factors, although the EP energy, which can be used to approximate the marginal likelihood, is non-increasing throughout the process.