Technical Report

UCAM-CL-TR-900
ISSN 1476-2986

Number 900

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

A framework to build bespoke auto-tuners with structured Bayesian optimisation

Valentin Dalibard

February 2017

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 2017 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. John's College.

ISSN 1476-2986

A framework to build bespoke auto-tuners with structured Bayesian optimisation

Valentin Dalibard

Summary

Due to their complexity, modern computer systems expose many configuration parameters which users must manually tune to maximise the performance of their applications. To relieve users of this burden, auto-tuning has emerged as an alternative in which a black-box optimiser iteratively evaluates configurations to find efficient ones. A popular auto-tuning technique is *Bayesian optimisation*, which uses the results to incrementally build a probabilistic model of the impact of the parameters on performance. This allows the optimisation to quickly focus on efficient regions of the configuration space. Unfortunately, for many computer systems, either the configuration space is too large to develop a good model, or the time to evaluate performance is too long to be executed many times.

In this dissertation, I argue that by extracting a small amount of domain specific knowledge about a system, it is possible to build a bespoke auto-tuner with significantly better performance than its off-the-shelf counterparts. This could be performed, for example, by a system engineer who has a good understanding of the underlying system behaviour and wants to provide performance portability. This dissertation presents BOAT, a framework to build **B**esp**O**ke **A**uto-**T**uners. BOAT offers a novel set of abstractions designed to make the exposition of domain knowledge easy and intuitive.

First, I introduce *Structured Bayesian Optimisation* (SBO), an extension of the Bayesian optimisation algorithm. SBO can leverage a bespoke probabilistic model of the system's behaviour, provided by the system engineer, to rapidly converge to high performance configurations. The model can benefit from observing many runtime measurements per evaluation of the system, akin to the use of profilers.

Second, I present *Probabilistic-C++* a lightweight, high performance probabilistic programming library. It allows users to declare a probabilistic models of their system's behaviour and expose it to an SBO. Probabilistic programming is a recent tool from the Machine Learning community making the declaration of structured probabilistic models intuitive.

Third, I present a new *optimisation scheduling abstraction* which offers a structured way to express optimisations which themselves execute other optimisations. For example, this is useful to express Bayesian optimisations, which each iteration execute a numerical optimisation. Furthermore, it allows users to easily implement *decompositions* which exploit the loose coupling of subsets of the configuration parameters to optimise them almost independently.

Acknowledgements

Foremost, I would like to thank my supervisor, Eiko Yoneki, for her constant support and encouragement over the past four years. Through countless meetings and discussions, Eiko has relentlessly encouraged me to strive for better results whilst never losing sight of the bigger picture. I am also grateful to my PhD examiners Zoubin Ghahramani and Willy Zwaenepoel for their insightful comments and suggestions.

The project at the centre of this dissertation, BOAT, has benefited from the feedback of many friends and colleagues. I am particularly indebted to Michael Schaarschmidt for being its first adopter and contributing to its evaluation. I thank Karthik Nilakant and Andre Ribeiro for fascinating discussions on the early versions of BOAT, and Ionel Gog, Malte Schwarzkopf and Tim Harris whose suggestions allowed me to greatly improve the quality of BOAT's exposition. I am also grateful to Michael Osborne for inviting me to give a talk in his research group which resulted in valuable feedback.

I thank the Computer Laboratory support staff and, in particular, Lise Gough for their friendliness, patience and constant assistance throughout my administrative incompetence. I am also grateful to EPSRC, the Computer Laboratory, St John's College and the Cambridge Philosophical Society for funding my travels and studies and giving me the opportunity to conduct my research in Cambridge.

My time in Cambridge was greatly enriched by the variety of projects I had the chance to partake in. I am grateful to Raoul-Gabriel Urma and the Cambridge Spark team for giving me the opportunity to work on refreshingly different computer science endeavours, and the members of the Cambridge University Fencing Club with whom I have shared so many experiences.

Finally, I am profoundly grateful to my family and friends whose love and counsel have carried me throughout this journey. I thank my parents for their encouragement and advice throughout my studies which I have valued immensely; my siblings, their families and Sandra and Stephen Kidd for their unending support; Zach Eaton-Rosen, whose relentless curiosity and rationality has always been a source of inspiration and Sarina Kidd – whose thorough proofreading accounts for the coherence of this dissertation – for accompanying me in this journey with wit and unwavering support.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

Auto-tuners are a class of tools designed to optimise the configuration parameters of computer systems in order to maximise their performance in a given context. This dissertation focuses on the construction of bespoke auto-tuners, specialised for a single system, which perform this optimisation in a short time compared to their off-the-shelf counterparts. In this introduction, I first discuss the growing demand for efficient auto-tuners due to the increase in complexity of modern systems. I then present my thesis, the contribution offered by this dissertation and outline the structure in which they will be presented.

Computer systems have become a key part of modern data processing pipelines. The role of a computer system is to provide an *abstraction*: users interact with the system in a way that is more declarative than the infrastructure that the system is built on top of. For example, this can mean abstracting a storage device to provide a file system, or a cluster of machines to expose a single computation engine. Over the years, systems have evolved to provide increasingly general abstractions, both in terms of the types of workloads they can execute and the hardware they can run on top of. While traditional databases were limited to performing simple relational queries, modern data processing systems can leverage distributed heterogeneous machines to execute complex iterative tasks.

This flexibility comes at a cost. Ideally, systems should provide *performance portability* and execute their assigned tasks efficiently, independently of the context in which they are executed. However, this variety of possible workloads or underlying hardware means there is no "one-size-fits-all" approach to best perform the execution. As a result, systems often expose *configuration parameters* which guide their behaviour and can be manually adapted to a user's context.

This manual tuning is a complex task, especially for non-experts users, as it requires an understanding of the underlying system behaviour. It breaks the notion of abstraction that systems are supposed to provide. Furthermore, users often run a stack of independent systems concurrently. In that case system configuration parameters will need to be adapted in synergy to obtain good performance, further increasing the difficulty of tuning. With the growth of complex computer systems, configuration tuning has become one of the main difficulties faced by users [BGO$^+$16].

Auto-tuning frameworks have been designed to tackle this issue and restore performance portability without the need for manual tuning. At the heart of an auto-tuner is an optimisation which iteratively evaluates the empirical performance of specific configurations to converge towards efficient ones. For example, Bayesian optimisation is a method often used for auto-tuning. It uses the measured results to incrementally build a probabilistic model of the impact of the parameters on performance. Unfortunately, for many complex computer systems, either the configuration space is too large to develop a good model, or the time needed to evaluate performance is too long to be executed many times. For these reasons, it is rare to see auto-tuners applied, for example, to distributed systems configurations.

This dissertation focuses on the construction of bespoke auto-tuners for computer systems. I propose to tackle the limitations of standard auto-tuners by exploiting *domain specific structure*. I have developed **BOAT**, a framework to build **B**esp**O**ke **A**uto-**T**uners. BOAT provides a set of abstractions to expose this structure and build a bespoke auto-tuner with better convergence than its off-the-shelf counterparts. This can be done, for example, by a system developer who has a good understanding of their system and wants to guarantee its performance portability. I use BOAT to argue the following thesis:

> Some black-box optimisation problems, such as tuning a program's parameters for performance, can significantly benefit from leveraging a small amount of domain specific information. The resulting optimisation performance can surpass that of off-the-shelf auto-tuners, making auto-tuning applicable in previously unexplored contexts.

1.1 Contributions

In this dissertation, I make three principal contributions, each of which is a key aspect of BOAT's architecture:

1. My first contribution is **Structured Bayesian Optimisation** (SBO), an extension of the Bayesian optimisation algorithm often used for auto-tuning. SBO exploits a user-given probabilistic model of the behaviour of the system being optimised to run an informed search on the configuration space. User-given probabilistic models can leverage multiple measurements per run of the system, akin to the use of profilers, to quickly infer the system's behaviour. Exposing an accurate model to an SBO is the key method proposed in this dissertation to exploit domain specific structure. I show how, by using SBO, one can construct optimisers that perform significantly better the traditional Bayesian optimisation algorithm.

2. My second contribution is **Probabilistic-C++**, a lightweight, high performance probabilistic programming library which can be used to easily declare arbitrary probabilistic models and perform full Bayesian inference on them. In the context of SBO, Probabilistic-C++ allows users to declare models of their system's behaviour. Performing inference on complex probabilistic models can be computationally expensive. Probabilistic-C++ offers ways to expose the probabilistic *independence* and *conditional independence* of a model to improve its convergence. Probabilistic-C++ also comes with a set of useful non-parametric models, including a novel treed Gaussian process implementation that is well suited to Probabilistic-C++'s inference algorithm.

3. My third contribution is **BOAT's optimisation scheduling abstraction**, which offers a structured way to express optimisations which themselves execute other optimisations. For example, this is useful to express Bayesian optimisations, which each iteration executes a numerical optimisation. Furthermore, these numerical optimisations are typically performed by an off-the-shelf optimiser. In large configuration spaces they may fail to converge due to the *curse of dimensionality.* BOAT's optimisation scheduling abstraction also allows a user to easily implement *decompositions* in which loosely related regions of the configuration space are optimised almost independently, improving the optimisation's convergence. Decompositions methods are a known and extensively used approach in optimisation. I present a spectrum of decomposition based techniques that can be used, the most advanced of which are inspired by reinforcement learning algorithms.

1.2 Dissertation outline

The rest of this dissertation is structured as follows:

Chapter 2 presents the Machine Learning techniques that BOAT is based on. In particular I focus on Gaussian Processes, an important class of non-parametric Bayesian models and show their use in the context of the Bayesian optimisation algorithm.

Chapter 3 introduces Structured Bayesian Optimisation (SBO) which is the key method I present to exploit domain structure. It also introduces the BOAT framework and presents a novel way to declare complex parameter spaces, including Turing-complete dependencies between a parameter's value and other parameters' existence.

Chapter 4 presents Probabilistic-C++ and its underlying inference algorithm. I show how independent parts of a model can be decoupled to help the model converge. I also discuss three topics which have proved useful in the context of modelling computer program's behaviour: *(i)* semi-parametric models which offer a simple approach for a developer to expose domain specific structure, while still accurately interpolating observed measurements, *(ii)* inference with non-Gaussian noise which is typically the case in the context of computer programs and *(iii)* non-parametric models which can scale to large numbers of measurements, in the form of treed-Gaussian Processes.

Chapter 5 is concerned with the numerical optimisation stage of Bayesian optimisation. I present BOAT's optimisation scheduling abstraction and how it can be used to implement decompositions of the configuration space. I also introduce two new techniques for decompositions, one based on Bayesian optimisation, the other inspired by reinforcement learning algorithms.

Chapter 6 combines the techniques introduced in the previous three chapters to show how to build a bespoke auto-tuner in BOAT using SBO. I also discuss the use of cheap experiments which can bring information about the behaviour of the system at a small cost.

Chapter 7 evaluates the application of BOAT and SBO via three case studies: *(i)* a garbage collection case study in which I tune the configuration flags of a database to minimise its 99th percentile latency, *(ii)* a sort case study in which I tune the implementation of a `sort` function to minimise its runtime based on the hardware and the input distribution of the arrays and *(iii)* a neural network case study in which I tune the scheduling of the training of a neural network on a distributed cluster to minimise its runtime. In each case study, I present the structure that was exploited to construct a bespoke auto-tuner and I quantify the improvements over off-the-shelf auto-tuners.

Chapter 8 presents related work, including auto-tuning frameworks and applications of modelling techniques to computer programs

Chapter 9 concludes this dissertation and outlines the directions for future work.

1.3 Related publications

Parts of the work described in this dissertation has been covered in peer-reviewed publications:

[**DSY17**] Valentin Dalibard, Michael Schaarschmidt and Eiko Yoneki. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the ACM International Conference on World Wide Web (WWW)*, April 2017. To appear.

[**DSY16**] Valentin Dalibard, Michael Schaarschmidt and Eiko Yoneki. Tuning the scheduling of distributed stochastic gradient descent with Bayesian optimization. In *NIPS Workshop on Bayesian Optimization*, December 2016.

CHAPTER 2

Bayesian optimisation

In this dissertation, my primary contributions are an extension of the Bayesian optimisation framework so it can leverage domain specific information. Therefore, this chapter surveys the different aspects of the Bayesian optimisation framework.

I begin this chapter with a review of *Gaussian processes*, a powerful class of Bayesian models which define a distribution over continuous functions (Section 2.1). Gaussian processes are often used in the context of Bayesian optimisation as a probabilistic model of the objective function. I then present Bayesian optimisation and show some of its implementation choices (Section 2.2).

2.1 Gaussian Processes

Gaussian Processes (GPs) are a class of probabilistic models that define a distribution over continuous functions. Although they have recently received significant interest in machine learning, they have long been used in various other scientific fields. In geostatistics and physics, GPs have been used for interpolation under the name *Kringing*, after Danie G. Kringe who developed a similar method in his master's thesis in 1951 [Kri51]. The term was later coined by Matheron [Mat62] who developed the theoretical basis for the method. They were later introduced in the context of Bayesian machine learning by Neal [Nea94] and have since received significant contributions from this community [Ras06].

In this section, I first review the basic inference mechanisms used by Gaussian processes (Section 2.1.1). I then consider their computational cost which can be significant with large datasets (Section 2.1.2). Section 2.1.3 is concerned with the choice of the *covariance function*, a component of GPs which states how smooth we expect the function being modelled to be. Finally, I briefly survey some extensions of GPs (Section 2.1.4).

2.1.1 Inference with Gaussian processes

Gaussian processes define a distribution over function mapping $\mathcal{X} \to \mathcal{Y}$ where $\mathbf{x} \in \mathcal{X}$ is a multi-dimensional input and $y \in \mathcal{Y}$ a real valued output. It is completely defined by two functions:

- $m(\mathbf{x})$ the mean function at $\mathbf{x}$, and
- $k(\mathbf{x}, \mathbf{x}')$ the covariance function, also called kernel.

The resulting Gaussian process is defined as

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

The covariance function is used to describe how similar two points are. A popular choice of covariance function when dealing with D-Dimensional input data is the squared exponential:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{1}{2}\|\mathbf{x}_i - \mathbf{x}_j\|^2\right), \tag{2.1}$$

although there are alternatives. I discuss the choice of covariance function in Section 2.1.3. An important restriction is that it must be positive definite.

If we want to sample from the prior distribution defined by a Gaussian process, we first define a set of inputs $\{\mathbf{x}_{1:n}\}$ which we are interested in. The goal will be to sample a corresponding set of sampled outputs $\{\mathbf{y}_{1:n}\}$. We first compute the covariance matrix

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_t, \mathbf{x}_1) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}$$

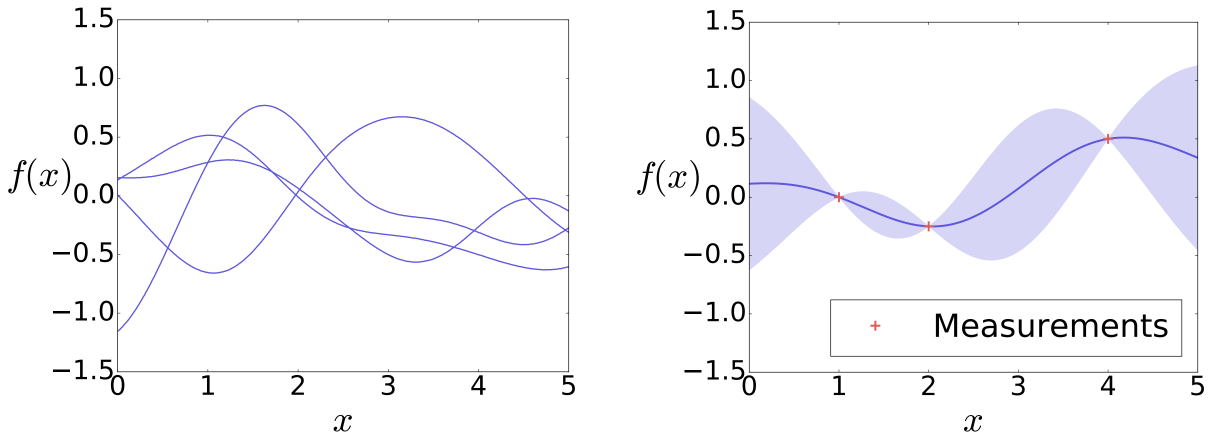
and the mean vector

$$\mathbf{m} = [m(\mathbf{x}_1) \cdots m(\mathbf{x}_n)].$$

Then, $\{\mathbf{y}_{1:n}\}$ can simply be sampled from the multivariate normal distribution $\mathbf{y}_{1:n} \sim \mathcal{N}(\mathbf{m}, \mathbf{K})$. For example, Figure 2.1a shows four samples drawn from a Gaussian process distribution.

In the context of Bayesian optimisation, we will rarely want to sample from the prior distribution. Instead, we will have a set of observed data $\{\mathbf{x}_{1:n}, \mathbf{y}_{1:n}\}$, and we will want to predict the posterior distribution of another input $\mathbf{x}_*$. From the prior distribution we know that

$$\begin{bmatrix} \mathbf{y}_{1:n} \\ y_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{m} \\ m(\mathbf{x}_*) \end{bmatrix}, \begin{bmatrix} \mathbf{K} & \mathbf{k}_* \\ \mathbf{k}_*^{\mathsf{T}} & k(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix}\right)$$



(a) Four samples from the prior distribution. **(b)** Posterior distribution after three measurements.

Figure 2.1: Example distributions of Gaussian processes.

where

$$\mathbf{k}_* = [k(\mathbf{x}_1, \mathbf{x}_*) \cdots k(\mathbf{x}_n, \mathbf{x}_*)].$$

The posterior distribution of y_* can be analytically solved [Ras06] as

$$p(y_* \mid \mathbf{x}_*, \{\mathbf{x}_{1:n}, \mathbf{y}_{1:n}\}) = \mathcal{N}\left(\mu(\mathbf{x}_*), \sigma^2(\mathbf{x}_*)\right)$$

where

$$\mu(\mathbf{x}_*) = m(\mathbf{x}_*) + \mathbf{k}_*^\mathsf{T}\mathbf{K}^{-1}(\mathbf{y}_{1:n} - \mathbf{m}) \tag{2.2}$$

$$\sigma^2(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^\mathsf{T}\mathbf{K}^{-1}\mathbf{k}_*. \tag{2.3}$$

For example, Figure 2.1b shows the mean and standard deviation of a Gaussian process after three measurements.

An important intuition is that, in a way, the GP never "sees" the true value of the inputs. Instead, it only sees how they relate to one another via the covariance matrix. Similarly, note that the mean vector can be moved out of the distribution definition: $\mathcal{N}(\mathbf{m}, \mathbf{K}) = \mathbf{m} + \mathcal{N}(\mathbf{0}, \mathbf{K})$. With this view, the GP also does not see the output values, only their differences from the mean function.

2.1.2 Computational complexity

I now discuss the computational complexity of three operations: adding a new observation to the model, predicting the mean value of an input, predicting the variance of an input.

Adding a new observation. The most expensive part of the above predicted distribution is the computation of the inverse of $\mathbf{K}$. If we have n points in our dataset, this has a cost $\mathcal{O}(n^3)$. In practice, we do not compute the actual inverse and instead compute the Cholesky decomposition of $\mathbf{K}$ which also has cost $\mathcal{O}(n^3)$.

In the context of Bayesian optimisation, we will often receive observations one after another. We can take advantage of the previously computed decomposition to reduce the complexity. Consider computing the Cholesky decomposition of the matrix

$$\begin{bmatrix} \mathbf{K}_{1:n} & \mathbf{k}_{n+1} \\ \mathbf{k}_{n+1}^\mathsf{T} & k(\mathbf{x}_{n+1}, \mathbf{x}_{n+1}) \end{bmatrix}$$

given the decomposition $\mathbf{L_{1:n}}$ of $\mathbf{K}_{1:n}$. Note that $\mathbf{L_{1:n}}$ is triangular. We can compute $\mathbf{l}_{n+1} = \mathbf{L}_{1:n}\backslash\mathbf{k}_{n+1}$ as the solution to the equation $\mathbf{L}_{1:n}\mathbf{l}_{n+1} = \mathbf{k}_{n+1}$ in $\mathcal{O}(n^2)$ steps. Then, the decomposition of $\mathbf{K}_{1:n}$ is simply

$$\mathbf{L}_{1:n+1} = \begin{bmatrix} \mathbf{L}_{1:n} & \mathbf{0} \\ \mathbf{l}_{n+1} & \sqrt{k(\mathbf{x}_{n+1}, \mathbf{x}_{n+1}) - \mathbf{l}_{n+1}^\mathsf{T}\mathbf{l}_{n+1}} \end{bmatrix}.$$

Hence, adding a new observation to the model can be computed in $\mathcal{O}(n^2)$.

Predicting the mean value of a new input. Equation 2.2 showed how to compute the mean prediction for an input $\mathbf{x}_*$. The term $\mathbf{K}^{-1}(\mathbf{y}_{1:n} - \mathbf{m})$ requires using the decomposition $\mathbf{L}$ of $\mathbf{K}$ and computing $\mathbf{L}\backslash(\mathbf{y}_{1:n} - \mathbf{m})$ at computational cost $\mathcal{O}(n^2)$. However, this is independent of $\mathbf{x}_*$. Hence, we can compute it once when adding an observation and reuse the computed value afterwards. The rest of the computation involves performing an inner product with $\mathbf{k}_*$ which can be done in $\mathcal{O}(n)$. Therefore, the computational complexity of predicting the mean value of an input is $\mathcal{O}(n)$.

Predicting the variance of a new input. Equation 2.3 involves computing $\mathbf{K}^{-1}\mathbf{k}_*$ which means solving $\mathbf{L}\backslash\mathbf{k}_*$. Unlike for the mean prediction, this is dependent on the value of $\mathbf{x}_*$ and hence must be performed per input. Hence, the complexity of computing the variance is $\mathcal{O}(n^2)$.

2.1.3 Choice of covariance functions

As mentioned in Section 2.1.1, a Gaussian process prior is fully defined by the mean and covariance functions. In Chapter 4, I will discuss *semi-parametric* models which will effectively allow a user to define a mean function. In this subsection, I consider the choice of covariance functions which specifies the smoothness of the GP's output. I discuss three properties: *hyperparameters* which are used to parametrise the covariance function and adapt it to the data, the use of noisy measurements, and the Matérn covariance function which is an alternative to the squared exponential defined in equation 2.1.

Hyperparameters. Often we will want to adapt the covariance function to the data. This is usually done via hyperparameters. Two types of hyperparameters are used: the *amplitude* hyperparameter σ_f^2, which states how far from the mean we expect to be, and the *length-scale* parameters $\boldsymbol{\ell}$, which squash or dilate the dimensions of the input space. If $\mathbf{x} \in \mathcal{X}$ id D-dimensional, then $\boldsymbol{\ell}$ is a D-dimensional vector $\ell_1 \dots \ell_D$ and the distance along the kth dimension of two inputs $\mathbf{x} = [x_1 \dots x_D]$ and $\mathbf{x}' = [x'_1 \dots x'_D]$ from the point of view of the covariance is $(x_k - x'_k)/\ell$.

The kernels I use are *isotropic*. Given the inputs $\mathbf{x}$ and $\mathbf{x}'$, they can be equivalently written as a function of the Euclidean distance $\|\mathbf{x} - \mathbf{x}'\|$. In the presence of the length-scale hyperparameters, the scaled Euclidean distance can be written as a function of the scaled square norm:

$$r^2(\mathbf{x}, \mathbf{x}') = \sum_{d=1}^{D} (x_d - x'_d)^2 / \ell_d^2.$$

For example, the squared exponential covariance function defined in equation 2.1 can be rewritten to include the hyperparameters:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-\frac{1}{2} r^2(\mathbf{x}_i, \mathbf{x}_j)\right).$$

The linear scales hyperparameters have a high impact on the behaviour of the GP and hence it is important to understand them well. Setting a high value of ℓ_i means all of the inputs will be squished along dimension i. This implicitly decreases the importance of that dimension. All points will be seen as similar in that aspect and discrimination will happen along other dimensions. On the other extreme, setting a very low value of ℓ_i means all of the points will be far from one another, ultimately leading for them to be viewed as uncorrelated by the GP. Within reasonable ranges, decreasing the value of ℓ_i means we expect functions to vary more along dimension i.

In practice, we do not know in advance the right values of the hyperparameters, we must therefore learn them from the data. This is usually done in one of two ways. One approach is to perform an optimisation and find the hyperparameter values which best explain the data. This is done by maximising the *marginal likelihood* (or *evidence*) of the Gaussian process with respect to the data. The second approach is to perform *Bayesian model selection* and place a prior distribution on the hyperparameters. We can then infer a posterior distribution of their values, usually as a set of samples.

When I use Gaussian processes in this dissertation, I always do so within the context of my probabilistic programming framework, described in Chapter 4. This allows me to use this second and more robust approach, I place priors on the hyperparameters and let the probabilistic engine perform inference on their values. However, this means I am not able to use more clever inference algorithms which are more suited to a Gaussian process's structure such as elliptical slice sampling [MAM10] or Hamiltonian Monte Carlo sampling [Nea11].

Noisy measurements. Often the measured values will be *noisy*. If we repeated the same experiment with the same input, we would obtain a slightly different output. Gaussian processes are able to model Gaussian noise. Say our measurements are distributed with noise variance σ_n^2. This can be added to the GP by adding an extra term to the covariance function when both inputs are from the same measurement:

$$k'(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} k(\mathbf{x}_i, \mathbf{x}_j) + \sigma_n^2 & \text{if } i = j \\ k(\mathbf{x}_i, \mathbf{x}_j) & \text{otherwise.} \end{cases}$$

We can even allow the noise variance to be a function of the input $\mathbf{x}$, which is called *heteroscedasticity.*

Matérn covariance function. In practice, the squared exponential covariance function tends to lead to functions that are too smooth to realistically represent the behaviour of computer programs. There are a range of alternatives. In this dissertation, I always use the Matérn 5/2 kernel as is recommended by Snoek et al. [SLA12]:

$$k_{M52}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \left(1 + \sqrt{5r^2(\mathbf{x}_i, \mathbf{x}_j)} + \frac{5}{3} r^2(\mathbf{x}_i, \mathbf{x}_j)\right) \exp\left(-\sqrt{5r^2(\mathbf{x}_i, \mathbf{x}_j)}\right).$$

2.1.4 Extensions to Gaussian processes

One of the major difficulties when dealing with Gaussian processes is their computational cost with large datasets. In this subsection, I highlight some of the extensions that have been proposed to allow them to scale.

Experts approaches. To reduce the computational burden, some approaches train multiple independent "experts" on different subsets of the data. *Mixture of experts* methods [JJNH91] have been applied to GP regression [MO06, RG01, YN09]. In this context, a gating network divides the domain of $\mathcal{X}$ into regions within which local experts make predictions. An advantage of this approach is that, since each GP is trained on a local dataset, we can infer the hyper-parameters of each GP independently, allowing them to adapt locally. Predictions for a new input $\mathbf{x}_*$ are made by querying each expert at $\mathbf{x}_*$ and weighing these predictions by responsibilities assigned by the gating network.

On the other hand, *product of experts* methods sidestep the need for weight assignment by training independent GPs whose predictions can be combined based on their confidence on the input [ND14, CF14, DN15].

Sparse approximations. Another alternative to reduce the computational cost of Gaussian process is to train a single GP on a reduced number of data points. This can be done either naively, by selecting a subset of the training data, or more accurately, by generating pseudo-inputs which capture the shape of the data [SWL03, SG05, Tit09]. The major limitation of these approaches, is the error in the variance estimates generated

Algorithm 2.1 The Bayesian optimisation algorithm

```
Input: Objective function $f()$
Input: Probabilistic model $G$
Input: Acquisition function $\alpha()$
1: for $t = 1, 2, \ldots$ do
2:     Sample point: $\mathbf{x}_t \leftarrow \arg\max_{\mathbf{x}} \alpha(\mathbf{x} \mid G)$
3:     Evaluate new point: $y_t \leftarrow f(\mathbf{x}_t)$
4:     Update probabilistic model: $G \leftarrow G \mid (\mathbf{x}_t, y_t)$
5: end for
```

by these methods. Predictions near pseudo-measurements will be over-confident. On the other hand, predictions at inputs from the dataset may have high variance.

Treed Gaussian processes. Similarly to expert approaches, treed GPs train multiple GPs on different subsets of the data [GL07]. Unlike them however, the predictions of the different GPs are not combined. Treed GPs are constructed in a similar way to the decision trees used by random forests for regression. The trees are binary trees, and each branch of the tree queries the aspect of an input. A branch b specifies a dimension u_b, along which to query the input, and a threshold value s_b. A prediction for an input $\mathbf{x}_*$ is made by propagating $\mathbf{x}_*$ down the tree. At each branch b, the u_bth dimension of $\mathbf{x}_*$ is compared with s_b. If it is greater, the input is propagated to the right sub-tree, otherwise, it is propagated to the left sub-tree. At the leaf of the tree, a Gaussian process that was trained using all data points in the training set that followed this path returns its prediction for $\mathbf{x}_*$. In order to average over possible tree branch values and structure, MCMC is used to sample different decision trees. At prediction time, we can combine the predictions from each sampled tree.

2.2 Bayesian optimisation

This section presents Bayesian optimisation, a methodology to find the extremum of an expensive black-box function. Although first introduced by Močkus et al. [MTŽ78] in 1978, it was only years later, once it became coupled with Gaussian processes, that the framework gained popularity.

I first present the Bayesian optimisation methodology (Section 2.2.1). Section 2.2.2 is concerned with the choice of acquisition function, which trades-off the exploration and exploitation of the procedure. Finally, I consider the limitations of Bayesian optimisation (Section 2.2.3).

2.2.1 Methodology

In its most classical setting, Bayesian optimisation attempts to find the maximum of some objective function $f(x)$ where $x \in \mathbb{R}^D$. To this end, it incrementally builds a probabilistic model which reflects the current knowledge of the objective function. Most of the time, a Gaussian process (GP) is used although random forests [Bre01] and neural networks have also been considered [HHLB11, SRS$^+$15].

Algorithm 2.1 shows the procedure. In each iteration, the optimisation executes three steps. First, it performs a numerical optimisation to find a point in the configuration space which maximises an *acquisition function*. Acquisition functions are designed to maximise a combination of exploration and exploitation. Given a point $\mathbf{x}$, they measure its quality based on the distribution predicted by the model at $\mathbf{x}$. I discuss them in details in the next subsection.

Finding a point which maximises the acquisition function can be performed using an off-the-shelf numerical optimisation algorithm. The implicit assumption is that getting a prediction from the model is orders of magnitude faster than executing $f()$, and hence we can afford to use a simpler optimisation algorithm. Because the model represents the objective function $f()$, which may be multimodal, we must use an algorithm which supports black-box global optimisation.

The DIRECT algorithm, which recursively sub-DIvides the space into sub-RECTangles is often used for this task [JPS93]. Another popular alternative is the CMA-ES algorithm [HO01], which is based on evolutionary methods.

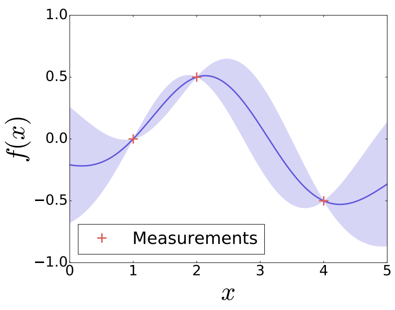
Second, the optimisation evaluates the expensive objective function at the point $\mathbf{x}_t$ found by the numerical optimisation. Third, it includes this new measurement into the model.

When compared with other optimisation methods, such as evolutionary algorithms, Bayesian optimisation tends to converge in fewer iterations. This however comes at the cost of a high overhead per iteration due to the computational cost of performing the numerical optimisation.

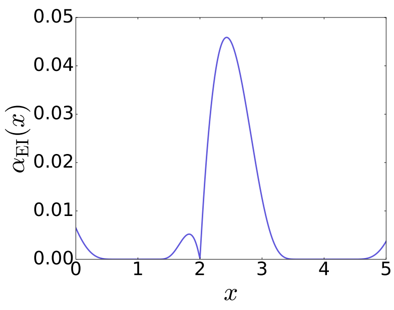
2.2.2 Acquisition functions

The goal of the acquisition function is to evaluate the goodness of an input $\mathbf{x}$ based on the model's prediction. We want to trade-off *exploitation*, evaluating configurations which we know will perform well, and *exploration*, evaluating configurations which will be informative about the shape of the objective function. This subsection lists a range of popular acquisition function. I follow the notation of Shahriari et al. [SSW$^+$16].

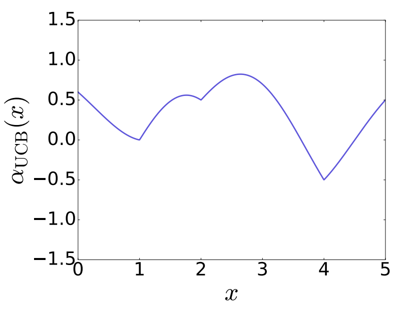
Possibly the simplest form of acquisition function is Thompson sampling [Tho33], originally introduced in the context of Bernoulli *bandit* problems (see [Sco10] for a complete



(a) Model distribution.



(b) Expected improvement.



(c) Upper confidence bound with $\kappa = 2$.

Figure 2.2: Examples of acquisition functions.

treatment). We simply sample a single model $g()$ from the current distribution of possible models G. Then the acquisition function is:

$$\alpha_{\mathrm{TS}}(\mathbf{x} \mid g) = g(\mathbf{x}).$$

If the numerical optimisation stage of the Bayesian optimisation finds the optimal value of $\mathbf{x}$ for $\alpha_{\mathrm{TS}}()$ then we are implicitly sampling from the distribution of optimal configurations of G. Until recently, this approach was not applied to Bayesian optimisation as it was not clear how to sample a single continuous function from a Gaussian process at an acceptable computational cost. However, spectral sampling techniques have recently been used to draw an approximate sample form the posterior [HLHG14], I discuss this approach in more details later in this subsection.

A popular acquisition function is the *expected improvement* which was originally proposed by Močkus et al. [MTŽ78]. For an input $\mathbf{x}$, it returns the expected value of the improvement brought by evaluating $f(\mathbf{x})$ over the best objective function value found so far η, also called *incumbent*:

$$\alpha_{\mathrm{EI}}(\mathbf{x} \mid \eta, G) = \int \max(0, g(\mathbf{x}) - \eta)\, p(g(\mathbf{x}) \mid G)\, dg(\mathbf{x}).$$

A useful feature of the expected improvement is that it has a closed form formula when the predicted distribution of $g(\mathbf{x})$ is normally distributed. If we are using a Gaussian process as the model, then given $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ the posterior mean and standard deviation at $\mathbf{x}$, the resulting expected improvement is:

$$\alpha_{\mathrm{EI}}(\mathbf{x} \mid \mu(\mathbf{x}), \sigma(\mathbf{x})) = \begin{cases} (\mu(\mathbf{x}) - \eta)\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & \text{if } \sigma(\mathbf{x}) > 0 \\ 0 & \text{if } \sigma(\mathbf{x}) = 0 \end{cases}$$

$$Z = \frac{\mu(\mathbf{x}) - \eta}{\sigma(\mathbf{x})}$$

where $\Phi(Z)$ and $\phi(Z)$ are the PDF and CDF of a standard normal distribution respectively. Figure 2.2b shows an example of the expected improvement function.

An alternative is the *upper confidence bound* (UCB), or lower confidence bound (LCB) when considering minimisation [SKSK10]. The acquisition function uses a positive parameter κ which explicitly trades-off the exploration and exploitation. Given the posterior mean $\mu(\mathbf{x})$ and standard deviation $\sigma(\mathbf{x})$ of an input $\mathbf{x}$, it is written as:

$$\alpha_{\text{UCB}}(\mathbf{x} \mid \mu(\mathbf{x}), \sigma(\mathbf{x})) = \mu(\mathbf{x}) + \kappa\sigma(\mathbf{x}).$$

Figure 2.2c shows an example of the upper confidence bound function.

Another recent promising avenue are acquisition functions which maximise the information gain about the distribution of the optimum brought by evaluating $\mathbf{x}$. *Entropy search* (ES) attempts to maximise

$$\alpha_{\text{ES}}(\mathbf{x} \mid G) = H(\mathbf{x}^* \mid G) - \mathbb{E}_{y|\mathbf{x},G} H(\mathbf{x}^* \mid (G \mid (\mathbf{x}, y)))$$

where $H(\mathbf{x}^* \mid G)$ is the differential entropy of the distribution of $\mathbf{x}^*$. The expectation is over the distribution of the random variable $y \sim G(\mathbf{x})$. Unfortunately, this function is not tractable for continuous spaces. Hence, approaches which use it rely on discretising the space $\mathcal{X}$ [VVW09, HS12]. An alternative is *predictive entropy search* (PES) which remove the need for this discretisation [HLHG14]. It leverages the symmetric property of mutual information to reformulate ES in terms of the differential entropy of the predictive distribution:

$$\alpha_{\text{PES}}(\mathbf{x} \mid G) = H(y \mid G, \mathbf{x}) - \mathbb{E}_{\mathbf{x}^*|G}[H(y \mid G, \mathbf{x}, \mathbf{x}^*)].$$

The right term requires averaging over the current distribution of optimums. This can be estimated through Thomson sampling. In the context of Gaussian processes, this can be done using spectral sampling techniques which approximate samples from the posterior distribution. Then, using simplifying assumptions, an approximation for the differential entropy $H(y \mid G, \mathbf{x}, \mathbf{x}^*)$ can be derived.

Portfolio strategies. It has been observed that no single acquisition function performs best over all problems, or even throughout the optimisation. Portfolio strategies use multiple acquisition functions concurrently. Each iteration, each acquisition function provides a candidate point to evaluate. A meta-criterion is then used to select among these candidates. Hoffman et al. keep track of the past performance of each acquisition function to select which should be used next [HBdF11]. More recently Shahriari et al. discriminate between candidates by measuring their information gain, in a similar way to entropy search techniques discussed above [SWH$^+$14]. Their formulation also requires a set of Thomspon samples, they generate them using the same spectral sampling methodology for Gaussian processes as used by Hernández-Lobato et al. [HLHG14].

2.2.3 Limitations

Recently, Bayesian optimisation has successfully been used to tune the hyper-parameters of machine learning applications and surpass human experts [SLA12, DSH13]. However, it has so far been unsuccessful at tackling optimisations in high dimensional spaces (more than 10) [SSW$^+$16]. There are two issues that may occur and limit the performance of a Bayesian optimisation:

1) The probabilistic model fails to accurately capture the objective function landscape after a reasonable number of iterations. This is due to the *curse of dimensionality* [Mur12]. The number of regions being modelled will grow exponentially with the number of dimensions. Hence, in order to converge in high dimensional domains, the model will require large numbers of measurements.

2) The numerical optimisation algorithm, used in each iteration, fails to converge and find a point with good acquisition function value. Once again, this can be linked to the curse of dimensionality. As the number of dimensions grows, the numerical optimisation will need to search an exponentially large number of regions.

In the rest of this dissertation I propose to extend Bayesian optimisation in a number of ways. The goal is always to provide abstractions allowing a user to tackle either of the above two issues.

2.3 Summary

This chapter focused on two related research research areas. First, I discussed Gaussian processes, a class of probabilistic model for regression. I highlighted their computational complexity which prevents their application to large datasets, and discussed some of the extensions that had been proposed to tackle this difficulty.

I then presented Bayesian optimisation, which most often uses a Gaussian process to incrementally model the objective function. I listed different approaches to select the next point to evaluate using the model's predictions. I discussed the limitations of Bayesian optimisation in high dimensional settings.

In the next chapter, I will present an extension of the Bayesian optimisation methodology which leverages the domain structure of an optimisation problem to tackle these limitations.

CHAPTER 3

Bespoke auto-tuners with structured Bayesian optimisation

In this chapter, I present the overall techniques that I use to create bespoke auto-tuners. I make the following contributions:

- I present Structured Bayesian optimisation(SBO), an extension of the Bayesian optimisation algorithm which leverages a user-given probabilistic model to quickly converge towards the objective function (Section 3.2).
- I introduce BOAT, my framework to build **B**esp**O**ke **A**uto-**T**uners (Section 3.3.1). BOAT includes a range of tools which can be used by a developer to construct an SBO.
- I present BOAT's configuration space abstraction, which is used to define the domain of an optimisation (Section 3.3.2). It allows developers to specify complex dependencies between parameters. In particular, the existence of a set of parameters can be made dependent on the value of another parameter.

Before this, Section 3.1 discusses the motivation for this dissertation. In particular, it gives an overview of the three optimisation problems tackled in Chapter 7, which I will use as running examples throughout the dissertation. It also lists a range of generic optimisation techniques which a developer would ideally be able to use in their bespoke auto-tuner.

3.1 Motivation

In this section, I first present the case studies evaluated in Chapter 7. These will both expose the necessity of having performant auto-tuners and serve as running examples throughout the rest of this dissertation. I then outline the different types of domain specific structures which can be taken advantage of by an optimisation.

3.1.1 Case Studies

To evaluate BOAT, I constructed three case studies designed to show a variety of contexts where auto-tuning is beneficial. The first case study, on garbage collection, only optimised three parameters. It showed that, even for simple optimisation problems, building a bespoke auto-tuner can yield significant convergence improvements. The second case study was concerned with optimising a sorting routine. It had over thirty parameters, a context in which off-the shelf auto-tuners are not applicable. It was designed show the applications of the optimisation methods presented here to complex parameter spaces with many dependencies between parameters. Finally, the third case study optimised the distributed scheduling of a neural network. It also had over thirty parameters. It was built to show that the methods presented here can be applied to complex distributed systems.

Garbage collection. The goal of this case study was to tune the garbage collection (GC) flags of a Java Virtual Machine (JVM) based database to minimise its 99th percentile latency. I tuned three parameters of the Concurrent Mark Sweep (CMS) collector: the *young generation size* and *survivor ratio* flags, which govern the size of the different sections of the heap, and the *max tenuring threshold* which sets the rate at which objects are promoted between heap sections [YL96]. I measured the 99th percentile latency of Cassandra [Apa16], a popular JVM-based wide-column store, using the YCSB cloud benchmarking framework [CST$^+$10] with a variety of workloads.

The behaviour of the garbage collection has a high impact on the latency. In one context, setting appropriate values of these three flags reduced the 99th percentile latency from 19ms – using Cassandra's default values – to 7ms. This is mostly due to *minor collections* which frequently collect objects in the young generation section of the heap, provoking a "stop-the-world" pause which halts the application. Good configurations will minimise the average duration of these collections as well as their total time. For example, Cassandra's default configuration uses a low young generation size, 100 MB per core. In practice this makes minor collections short but frequent, taking a large fraction of the total time. Increasing the young generation size will lead to more batching and will improve performance, but only up to a point, as large values will lead to long pauses raising latency again.

The small domain of this tuning problem means that off-the-shelf auto-tuners are still applicable. In my evaluation, I found that Spearmint [SLA12] converged to good configurations in 16 iterations after four hours of auto-tuning time (15 minutes per evaluation). However, exploiting contextual information can significantly reduce convergence time.

My auto-tuner, implemented in BOAT, is simple yet converged to within 10% of the best found performance by the second iteration. The next two case studies tackle more complex tuning problems – each with over thirty dimensions – in which off-the-shelf auto-tuners fail to find good values after thirty iterations.

Sort. In this case study, I tuned the implementation of a sorting procedure based on the distribution of input arrays and the underlying hardware. The optimisation domain is based on the implementation of `std::sort` used by `libstdc++`, `gcc`'s standard library. `std::sort` uses a hybrid implementation of *quicksort* and *insertion sort*. Quicksort is recursively used on the array, up to the point where the length of the sub-arrays are smaller than a single parameter called `block_size`, after which it switches to insertion sort. As noted by Ansel et al. [ACW+09], the default value of `block_size`, 32, is better suited to older hardware architectures. Furthermore, insertion sort performs well on arrays which are almost sorted. Therefore, using larger values of `block_size` on almost sorted arrays leads to better performance.

I designed a decision tree configuration space which dynamically queries an array's properties to dispatch it to an appropriate `block_size` parameter value. Branches of the decision tree scan the first elements of the input array and count the number of pairwise unsorted elements. This is used to estimate how sorted the array is. Counting the number of unsorted elements has a cost and hence there is a trade-off between the quality of the dispatch and the querying cost. This, along with the selection of appropriate `block_size` values at the leaves of the tree, is what is optimised by my auto-tuner.

The decision trees generated by my bespoke auto-tuner were up to four times faster on average than `std::sort`'s implementation. Furthermore, off-the-shelf auto-tuners were unable to cope with the large parameter space. When executed for sixty iterations, their median optimised configuration was almost twice slower than ones found by the bespoke auto-tuner after twenty iterations.

Neural network scheduling. In this case study, I tuned the distributed scheduling of the training of a neural network onto a heterogeneous cluster. There are two major components of this configuration space.

First, the computation load must be balanced across workers. Stochastic gradient descent, used to train neural networks, involves getting estimates of the gradient using many inputs in parallel. Hence, the auto-tuner must find how many inputs should be allocated to each worker based on their computational speed.

Second, the set of workers must be selected. A synchronisation barrier at each iteration of stochastic gradient descent makes all workers share and update their weights. The amount of data exchanged grows linearly with the number of workers and hence the slowest machines should not be used at all. I optimised this scheduling for a variety of clusters and neural network architectures.

	Model convergence	Numerical optimisation convergence
Embedding in the configuration space	✓	✓
Structured probabilistic model (§3.2, 4)	✓	
Many runtime measurements (§3.2, 4)	✓	
Decomposition methods (§5.3 - 5.6)		✓
Exploratory experiments (§6.2 - 6.3)	✓	
Cheap experiments (§6.4)	✓	

Table 3.1: List of techniques to improve an optimisation's convergence.

In my experiments, the optimised configurations found by my bespoke auto-tuner were up to 2.9× faster than simple configurations a user may have selected. Once again, the large dimensionality of the space prevented off-the-shelf auto-tuners from converging. Their median time after thirty iterations were over twice slower than the ones found by the bespoke auto-tuner after ten iterations.

Although these case studies are diverse, I used the same set of generic techniques to build each bespoke auto-tuner. The next subsection lists these techniques and discusses their impact.

3.1.2 Techniques to exploit

This subsection discusses the different techniques that we would like to exploit when optimising a computer system's configuration. Recall from Section 2.2.3 that there are two issues that can limit the performance of a Bayesian optimisation. Either the model used fails to converge after a reasonable number of iterations, or the numerical optimisation, performed each iteration, fails to find configurations viewed as promising by the model. Each optimisation technique presented in this dissertation is designed to combat one or both of these issues. I list them in table 3.1 and discuss each in turn.

Embeddings. Using an *embedding* means performing the optimisation over a subspace of the configuration space which we know contains the optimum. This reduces the size of the domain that has to be searched by the optimisation. For example, say we were optimising the performance of an application running two concurrent processes on a machine with eight cores. If two parameters of the configuration space specified the number of threads used by each process, we may start by allowing these parameters to be any positive integer. However, by reasoning about the parallelism of the machine, and the cost of context switches we may decide that the processes should use a total sum of eight threads, leading to only seven possible configurations.

Although simple, this kind of structure is crucial to reduce the complexity of the optimisation and should be employed whenever possible. It does not benefit from abstractions and hence I do not discuss it in the technical part of this dissertation. I do, however, mention the embeddings that were exploited in my case studies.

Structured models and multiple measurement. To help the Bayesian optimisation model converge, we can exploit some of the understanding we have of the objective function landscape. This is what I propose in the next section by using a user-given *structured probabilistic model*, and is the core idea of structured Bayesian optimisation. Another advantage of using such bespoke models is that they can exploit *many runtime measurements*, allowing them to converge even faster.

Decompositions. *Decomposition methods* are well known techniques which improve the convergence of an optimisation by exploiting some of its independences. I will show how they can be exploited to help the convergence of the numerical optimisation stage of a Bayesian optimisation.

Exploratory experiments. In order to explore the domain of the optimisation, Bayesian optimisation uses acquisition functions (Section 2.2.2), which trade-off exploration and exploitation. Exploratory experiments target regions of the domain with high uncertainty. It is the role of the numerical optimisation to find a configuration with high acquisition function value. In practice, this can be difficult. Due to their as acquisition functions are often highly multi-modal. Furthermore, their structure is typically not well suited to the use of decomposition methods. I will present techniques which tackle both issues.

Cheap experiments. Finally, *cheap experiments* are experiments that give us some information about the landscape of the utility function at a fraction of its cost. We would like to be able to exploit them to help the Bayesian optimisation model converge at a quicker rate. I will suggest how this can be done in the context of BOAT.

To summarise, this section highlighted some contexts in which having an efficient auto-tuner would be useful. It then listed a variety of techniques which could be employed by an auto-tuner. Although these techniques are simple in concept, they may be difficult to employ when building a bespoke auto-tuner from scratch. Hence, I argue that there is a need for a framework which makes the use of these techniques easy and intuitive. The next section presents structured Bayesian optimisation, the core methodology which my framework BOAT is built upon.

3.2 Structured Bayesian optimisation

In this section, I present *Structured Bayesian Optimisation* (SBO). SBO extends the Bayesian optimisation methodology by using a user-given *structured probabilistic model* of the underlying function being optimised, instead of a generic model like a Gaussian process. In the context of creating a bespoke auto-tuner for a computer system, this

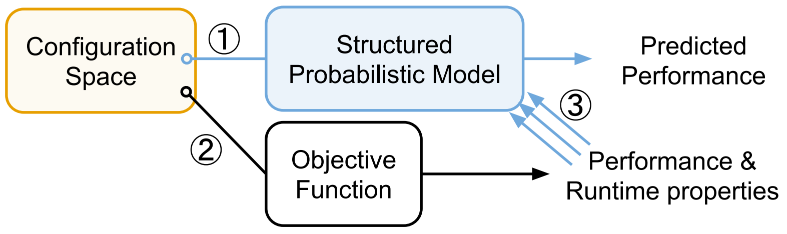

Configuration Space
①
Structured Probabilistic Model
Predicted Performance
③
②
Objective Function
Performance & Runtime properties

Figure 3.1: Procedure of structured Bayesian optimisation.

model would be implemented by a system developer that has a good understanding of the underlying system behaviour.

The procedure of an SBO is similar to the one of a traditional Bayesian optimisation and is presented in Figure 3.1. It performs three steps at each iteration: (1) it looks for a configuration with good predicted performance by the bespoke model. (2) It evaluates the best found configuration using the objective function and collects runtime measurements. (3) It performs inference on the model using all observations.

When compared with traditional Bayesian optimisation, using bespoke models brings three main advantages. First, it captures the user's understanding of the behaviour of the system. This drastically reduces the number of iterations needed for the model to converge towards the objective function. In the context of BOAT, models are implemented in probabilistic programming and can reason about arbitrary data structures – like regular programs – and reproduce complex behaviours. For example, the model in the neural network case study predicts the individual computation time of each machine in a distributed cluster. The total time is predicted to be the highest of these individual times plus a communication cost. It would take many evaluations for a generic probabilistic model to accurately model the function *max* over multiple inputs, but our model does so by default.

Second, it makes Bayesian optimisation applicable to new domains with complex configuration spaces, where the existence of some of the parameters may be conditional. If instead we were using Gaussian Processes, we would have to design a covariance function over this complex configuration space. This is a difficult task, recall from Section 2.1.1 that the covariance function must be positive definite. The construction of kernels for complex data structures, such as trees, is an active area of research. For example, Swersky et al. [SDS+13] recently derived a covariance function for configuration spaces with a fixed number of parameters, but where the relevance of some parameters was dependent on the value of others. However, the approach is not generalisable to recursive parameter spaces, such as trees, with a possibly infinite number of dimensions. On the other hand, probabilistic programs can reason about these complex configuration spaces and interpret them in a suitable way.

Third, using such a model allows us to collect many runtime properties reflected in the model and use them for inference. For instance, in the garbage collection case study, the

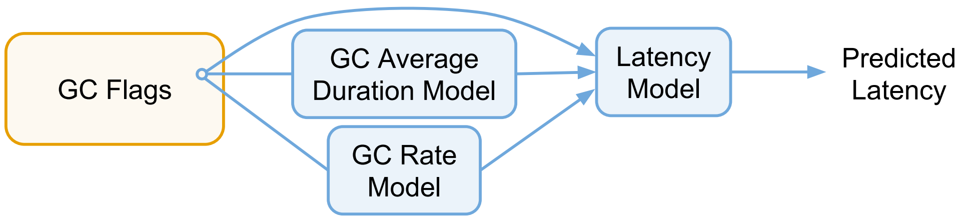


Figure 3.2: Dataflow of the garbage collection model

model predicts the rate and average duration of minor collections. After each experiment, the garbage collection logs are parsed to observe their true value of these statistics and use them for inference. In Chapter 6, I will further take advantage of this aspect to suggest how we can leverage cheap experiments, which can give some information about the behaviour of the system at a fraction of the cost of the objective function.

3.2.1 Bespoke probabilistic models

In this subsection, I illustrate the construction of a structured probabilistic model which captures the behaviour of the underlying system. I do so using the model used in the garbage collection case study.

In the context of BOAT, the probabilistic model written by the system developer should take as input a configuration and predict its performance. Initially, developers should use a generic model, effectively running traditional Bayesian optimisation, and observe whether the convergence time is acceptable. If it is not, SBO allows users to incrementally add *structure* to the model to reduce convergence time. Adding structure is done by making the probabilistic model more similar to the behaviour of the system.

In the garbage collection case study, I used a Gaussian process (GP) as the initial model. The GP predicted 99th percentile latencies based on the flag values. This took many iterations to converge, despite the simplicity of the problem. To add structure, I included in the model a notion of *rate* and *average duration* of minor collections. Given flag values, the model predicted both these statistics. It then predicted the latency as a function of the flag values and the statistics. The data flow of the model is shown in Figure 3.2.

When using the model in BOAT, I collected the true value of these statistics from the GC logs after each evaluation and use them for inference. Further, I declared how I believed each of the three model components behaved as a function of their inputs. For example, I noticed the rate of minor collections was inversely proportional to the size of the *eden* memory region in the JVM-heap, which is where objects are initially allocated. This intuition was included in the corresponding model by building a *semi-parametric model* (Section 4.3.1), which can successfully combine a user-defined trend with empirical data.

In practice, I find that even adding a little structure can be sufficient to make the optimisation converge in a reasonable time. This is useful as simpler models are able to adapt

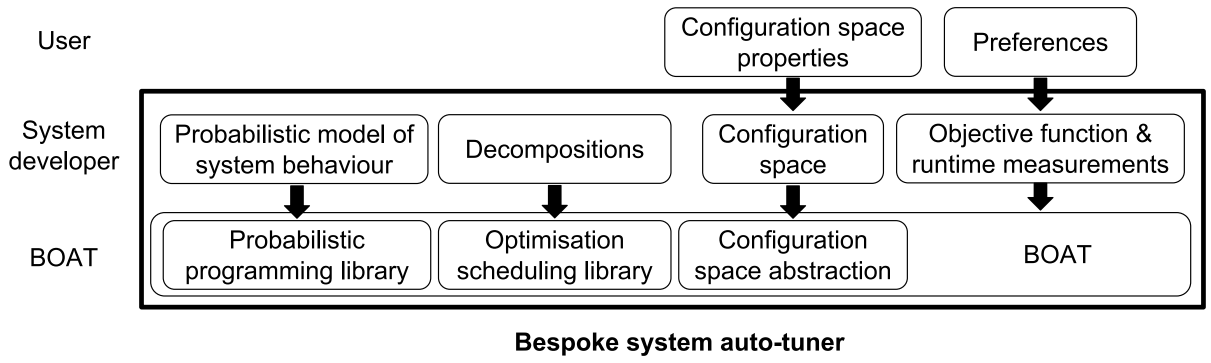

User
Configuration space properties
Preferences
System developer
Probabilistic model of system behaviour
Decompositions
Configuration space
Objective function & runtime measurements
BOAT
Probabilistic programming library
Optimisation scheduling library
Configuration space abstraction
BOAT
Bespoke system auto-tuner

Figure 3.3: Overview of the flow of data when using BOAT

to a broader variety of behaviours, such as widely different hardware and workloads. This allows for the construction of bespoke auto-tuners that provide global performance portability.

3.3 BOAT

This section presents BOAT, my framework to build bespoke auto-tuners. The goal of BOAT is to allow a system developer to design an auto-tuner with a good convergence rate for their system, hence guaranteeing their system's performance portability. I begin this section with an overview of BOAT and its use. I then present BOAT's configuration space API which allows the declaration of optimisation domains with dependencies between parameters.

3.3.1 Overview of BOAT

Figure 3.3 shows an overview of the flow of data when using BOAT. Application users provide two types of arguments specific to their application:

- **Configuration space properties**: These have an influence on the set of valid configurations. In a distributed scheduling problem, this could be the list of available machines.
- **Preferences**: These define system performance metrics. For example, a user could set the workload with which the system should be evaluated, or specify whether to optimise throughput or tail latency.

To create a bespoke auto-tuner, a system developer takes these arguments as input to provide four types of information to BOAT:

1) **Configuration space**: The space of valid configurations which should be explored. This is also called the *domain* of the optimisation.

2) **Objective function & runtime measurements**: This specifies how to evaluate a given configuration. For example, this can involve writing configuration values to a configuration file and starting a distributed system along with a benchmark.

3) **Probabilistic model of system behaviour**: The contextual information which allows BOAT to discard regions of low performance and quickly converge.

4) **Decompositions**: How to decompose the numerical optimisation so independent parameters can be optimised individually.

Note that the first two items specify the optimisation problem, while the latter two are designed to give structure to the optimisation to make it converge quickly. All of BOAT's components are implemented in C++.

3.3.2 BOAT's configuration space abstraction

Often when tuning programs, the existence of a parameter is dependent on the value of another. For example, when optimising a decision tree, each node can either be a branch or a leaf, and the existence of left and right sub-trees is dependent on that status. For this reason, the domain often used in numerical optimisation, $\mathcal{X} \subseteq \mathbb{R}^d$, is insufficient for our needs. In this section, I show how more complex configuration spaces are declared in BOAT.

I distinguish two related constructs to define domains of valid parameters. The first is traditional parameters which can take some scalar or categorical values. In BOAT, these are called `Parameter`'s. For example,

```
RangeParameter<int> x1(0,5);
```

declares a parameter which can take integer values in the range $[0, 5]$. If `x1` has been assigned a value, this can be queried as `x1.value()`.

The second are parameter functions, which are normal executable functions with the added construct of being able to read parameter values and construct new parameters. In BOAT, `ParameterSpace` objects are designed to hold `Parameters` and may have a parameter function to create `Parameters` whose existence is conditional.

Listing 3.1 shows the implementation of a generic configuration space for decision trees in BOAT. The class `Node` inherits from `ParameterSpace` and has some parameters as fields. The first is a `BoolParameter`, called `is_leaf`, specifying whether the node is a branch or a leaf. The remaining ones are templated `ParameterPtrs`. This allows for them to only

```
template <class Leaf, class Query>
struct Node : public ParameterSpace {

  void parameter_function() override {
    // The existence of the parameters is dependent on the value of is_leaf
    if (is_leaf.value()) {
      leaf.new_parameter();
    } else {
      query.new_parameter();
      left.new_parameter();
      right.new_parameter();
    }
  }

  // Fields
  BoolParameter is_leaf;
  ParameterPtr<Leaf> leaf;
  ParameterPtr<Query> query;
  ParameterPtr<Node> left;
  ParameterPtr<Node> right;
}
```

Listing 3.1: Example of a decision tree configuration space.

be constructed depending on the value of `is_leaf`. The `parameter_function` method, overridden from `ParameterSpace`, defines how this construction is executed.

By allowing the use of arbitrary C++ functions as parameter functions, BOAT allows the construction of complex parameter spaces with dependencies between parameters. The domain of a parameter can be dynamically restricted in a similar way. Arguments can be passed to the parameter's constructor via

```
parameter_ptr.new_parameter(args);
```

For example, say we wanted a `RangeParameter<int>`'s domain to be bounded by a value `max` dependent on the values of the other parameters. Then we could declare a field `ParameterPtr<RangeParameter<int>> param`. In the parameter function, we would then compute the value of `max` and use

```
param.new_parameter(0, max);
```

This configuration space interface decouples the optimisation procedure – assigning values to parameters – from the configuration space properties. In BOAT, the optimisation iteratively assigns values to parameters, such as `is_leaf`. The framework then automatically constructs other parameters which exist as a consequence of those value assignments. This allows the easy comparison of different optimisation procedures.

The sort case study configuration space. As an example, I here show a simplified version of the configuration space of the sort case study, originally introduced in Section

```cpp
struct SortQuery {
  SortQuery() : scan_length(0, 5000),
                threshold(0, 5000) {}
  RangeParameter<int> scan_length;
  RangeParameter<int> threshold;
};
struct SortLeaf {
  SortLeaf() : block_size(1, 1 << 30) {}
  RangeParameter<int> block_size;
};
typedef Node<SortQuery, SortLeaf> SortNode;
```

Listing 3.2: A simplified declaration of the sort case study parameter space.

```cpp
void opt_sort(std::vector<int>& input){
  int cumul = 0;
  cumul += count_unsorted(input, 0, 166);
  if (cumul > 30) {
    opt_sort(input, 6);
  } else {
    cumul += count_unsorted(input, 166, 579);
    if (cumul > 4) {
      opt_sort(input, 9);
    } else {
      opt_sort(input, 12);
  }
}
```

Listing 3.3: A sort configuration's implementation.

3.1.1. It consists of a decision tree whose queries scan the input array's first elements to establish how sorted it is. At the leaves, it selects a value for `block_size`, an algorithmic parameter of the sorting procedure. Listing 3.2 shows a simplified implementation which uses the `Node` definition from Listing 3.1. Queries have two parameters, `scan_length` and `threshold`, which can each take values from 0 to 5000. Leaves have a single parameter `block_size`. In practice, I also use two types of embeddings. First, I bound the maximum depth of the decision tree to avoid "overfitting" the arrays representing the input distribution. Second, I restrict `block_size` to have values that are powers of two.

When I implemented this configuration space, I also implemented an objective function which turned a configuration into C++ code, compiled it and evaluated its performance. As an example, Listing 3.3 shows a possible configuration's implementation. The variable `cumul` keeps track of the number of unordered elements observed from the root of the decision tree. Each leaf specifies a value for the log base two of `block_size`.

In conclusion, BOAT includes multiple components which can be used together to build bespoke auto-tuners. This section highlighted BOAT's configuration space abstraction which can be used to define complex recursive domains for optimisations.

3.4 Summary

In this chapter, I started by highlighting the need for a structured framework to perform optimisations. This was based around two observations. First, there are many implicit optimisation problems in systems which would benefit from an adequate auto-tuner. Second, there are multiple generic optimisation techniques which can be applied to a range of problems. This motivated the construction of BOAT, a structured framework to build bespoke auto-tuners.

I then presented structured Bayesian optimisation, the key optimisation approach introduced in this dissertation. SBO extends Bayesian optimisation by leveraging a structured probabilistic model of the function being optimised. I then introduced the BOAT framework which includes a number of abstractions to build a bespoke auto-tuner based on SBO.

Figure 3.3 showed the dataflow when constructing a bespoke auto-tuner. It listed three of BOAT's components:

Configuration space abstraction This was presented in Section 3.3.2. It allows the construction of configuration spaces with complex dependencies. Both the sort and neural network case studies make extensive use of this construct.

Probabilistic programming library This will be the topic of Chapter 4. It allows a system developer to define a probabilistic model for the behaviour of their system, which can then be exposed to BOAT.

Optimisation scheduling abstraction This will be the topic of Chapter 5. One of the main uses of this abstraction will be the implementation of decompositions to help the numerical optimisation converge.

Finally, Chapter 6 will discuss how these different components can be put together to build a bespoke auto-tuner.

CHAPTER 4

Probabilistic models of programs executions

A structured Bayesian optimisation leverages a bespoke probabilistic model of the objective function to quickly converge. This chapter discusses the construction of such models. I introduce a number of techniques and abstractions which allow a developer to easily implement a probabilistic model of their system's behaviour. I make the following contributions:

- I present the design and implementation of Probabilistic-C++, a lightweight and high performance probabilistic programming library (Section 4.2). Probabilistic programming is a recent framework for expressing probabilistic models. Probabilistic-C++'s implementation makes it over 1000$\times$ faster than similar state-of-the-art probabilistic programming languages. Beyond performance, it has two key features. *Incremental inference*: when a new item is added to a dataset, the previously inferred model distribution can be used to do the minimum amount of work necessary.*Exploiting independence*: the probabilistic independence of a model can be exposed to help the inference converge.
- I outline a methodology to build Probabilistic-C++ models in the context of an optimisation in BOAT (Section 4.3). I discuss *semi-parametric models* which combine a user defined parametric model with a non-parametric model, such as a Gaussian process. Semi-parametric models offer an intuitive methodology to construct models of a system's behaviour.
- I show how Probabilistic-C++'s inference mechanism is well suited to do inference on complex noise functions (Section 4.4). I present a data structure called `OSTimeSampler` which can be used to construct an accurate distribution of the noise in runtimes when executing brief, single threaded computer programs. I evaluate the gain in modelling power on a dataset of matrix multiplication runtimes.

- I present a treed Gaussian process data structure that is designed to tackle the high computational complexity of Gaussian processes and is well suited to Probabilistic-C++ (Section 4.5). This allows developers to build models which perform inference on many measurements without having to sacrifice the computational performance of the model. I evaluate the data structure on the matrix multiplication dataset as well as a dataset from the sort case study.

- I show a method to generate an approximate model sample from a Probabilistic-C++ model distribution. I will use this technique in subsequent chapters to generate approximate Thompson samples (Section 4.6).

Probabilistic-C++ is strongly inspired by the Probabilistic-C probabilistic programming language. It uses the same inference algorithm, which is based on a particle filtering method called sequential Monte Carlo (SMC). Section 4.1 reviews the concepts behind probabilistic programming and the use of SMC for inference on probabilistic programs.

4.1 Probabilistic Programming

This section presents the concepts behind probabilistic programming. Section 4.1.1 discusses the semantics of probabilistic programs and how they can be used to express probabilistic models. I then review sequential Monte Carlo, a probabilistic inference algorithm, and show how it can be used in the context of probabilistic programs (Section 4.1.2).

4.1.1 The semantics of probabilistic programs

Probabilistic programming is a recent tool from the Machine Learning community which generalises graphical models and makes the construction of structured probabilistic models intuitive. It is akin to regular programming with the following added constructs [GHNR14]:

1. the ability to draw values from random distributions,
2. the ability to constrain the value of a variable via observations, and
3. the ability to output the distribution of a variable.

Unlike regular programs which are written to be executed, probabilistic programs are written to specify a probability distribution. For example, graphical models [KF09], such as Bayesian networks, can be represented in probabilistic programming. The goal of

```
# Draw from distributions
bias = uniform_draw(0.0, 1.0)
flip = bernoulli_draw(bias)

# Observe an outcome
observe(flip, true)

# Output the resulting distribution
predict(bias)
```

Listing 4.1: A very simple probabilistic program.

probabilistic programming is to make the construction of such probability distributions easy, hiding away the difficulties involved with performing inference. In this dissertation, I use probabilistic programs to model the behaviour of computations.

For example, Listing 4.1 shows a very simple probabilistic program in Python-like pseudo code. First, it sample a value from the uniform distribution in the range $[0.0, 1.0]$ and assigns it to a variable `bias`. It then draws a value from a Bernoulli distribution with parameter `bias` and assigns it to a variable `flip`. That is, `flip` will be `true` with probability `bias` and `false` with probability `1.0 - bias`. The next statement performs an observation, it constrains the value of `flip` to be `true`. Finally, `predict(bias)` outputs the distribution of `bias`, in light of this observation. Intuitively, this should be centred on the upper side of the range $[0.0, 1.0]$.

One way to understand the semantics of a probabilistic program is as follows. Performing inference is equivalent to running the program many times as a traditional program, but only outputting a value at `predict` statements if, for all previous `observe` statements, both arguments had the same value. In our example, this means only outputting the value of `bias` if the value of the variable `flip` was exactly `true`. As a consequence, the output values of `bias` are ones that explain our observation well.

Relationship with the traditional machine learning approach In machine learning, users typically perform inference on a *probabilistic model*. To do so, they use a dataset containing many independent samples. Typically, this would be expressed in probabilistic programming in the following way. First, the model would be defined, sampling each model parameter. Then, each sample in the dataset would lead to one or multiple observations. Intuitively, this makes the parameters of the model agree with the data. Finally, either the inferred distribution of the model parameters, or their predictions on a test dataset, would be output.

For example, Listing 4.2 shows pseudocode implementing linear regression, with the use of probabilistic programming constructs highlighted. The linear regression model has three parameters, `alpha`, the slope, `beta`, the y-intercept and `noise`, the noise at each observation. The listing performs inference on `alpha` and `beta` while keeping `noise` fixed

```
# Define the two model parameters
alpha = normal_draw(0.0, 1.0)
beta = normal_draw(0.0, 1.0)
noise = 2.0

# Define the model
def linear_model(x):
  return normal_draw(alpha * x + beta, noise)

# Perform observation x=1.0, y=3.0
# Draw from the model
sample1 = linear_model(1.0)
# Constrain the sample
observe(sample1, 3.0)

# Perform observation x=3.0, y=5.0
sample2 = linear_model(3.0)
observe(sample2, 5.0)

# Output beta's distribution
predict(beta)
```

Listing 4.2: Linear regression pseudocode in Probabilistic Programming.

at 2.0.

It draws both `alpha` and `beta` from a normal distribution with mean 0.0 and variance 1.0. It then performs two observations, i.e. use a dataset of size two. Each constrains a prediction from the model to match an observed value. Intuitively, the values of the observations ($x = 1.0, y = 3.0$ and $x = 3.0, y = 5.0$) mean we should expect the values of `alpha` and `beta` to be on the upper range of their initial distribution. Finally, `predict(beta)` outputs the distribution of the `beta` parameter, typically as a set of samples.

In the linear regression example, we are performing inference over two continuous probabilistic variables: `alpha` and `beta`. This makes it amenable to a number of frameworks, such as BUGS [STB$^+$96] or Stan [CGH$^+$16] which perform inference on a restricted set of probabilistic programs. Recently, methods to perform inference on *Turing complete* probabilistic programs have been proposed [GMR$^+$08].

For example, Listing 4.3 shows the definition of a probabilistic program which includes a `while` loop. In it, we again perform inference on the bias of a biased coin. The `biased_coin_model` keeps flipping a coin with probability `bias` until it return `true`. Once finished, it returns the number of flips that were performed. We should expect a low number of flips to be generated by high values of `bias` and vice versa. Our dataset has one sample in it saying the coin was observed to produce its first “head” on the fifth flip. Hence, we should expect `bias` to have a low value.

```
# Define the model parameters
bias = uniform_draw(0.0, 1.0)

# Define the model
def biased_coin_model():
  count = 0
  flip_head = False
  while not flip_heap:
    count += 1
    flip_head = bernoulli_draw(bias)
  return count

# Draw from the model and constrain the sample
sample = biased_coin_model()
observe(sample, 5)

# Ouput the bias' distribution
predict(bias)
```

Listing 4.3: A probabilistic program with a while loop.

For a probabilistic language to have a `while` loop, it must be Turing-complete. For example, the probabilistic programming frameworks BUGS [STB+96], Stan [CGH+16] and Infer.NET [MWG+] are not Turing complete and would not represent the above program in its current form. Recently, Church [GMR+08] was proposed as the first Turing complete probabilistic programming language. Since then a number of similar frameworks have been proposed [TvdMW15, MSP14].

In Section 4.2, I will present Probabilistic-C++, a new lightweight and high performance framework for probabilistic programming. The next subsection describes, as a preliminary, the particle filtering algorithms which Probabilistic-C++ and some of the other frameworks are based on.

4.1.2 Inference on probabilistic programs

This subsection presents sequential Monte Carlo (SMC), a particle filtering technique used for inference in probabilistic programs [DDFG01]. First, I introduce *importance sampling* which is a key component of SMC. I follow the notation of MacKay [Mac03]. See Doucet and Johansen [DJ09] for a complete introduction to SMC methods.

Importance sampling. Importance sampling is used when we want to estimate the expectation of a function $\phi(\mathbf{x})$, where $\mathbf{x}$ is drawn from a distribution $P(\mathbf{x})$ which we cannot directly sample from. Usually we cannot evaluate the density $P(\mathbf{x})$ directly but instead have access to a function $P^*(\mathbf{x})$ which is identical up to a multiplicative constant:

$$P(\mathbf{x}) = P^*(\mathbf{x})/Z.$$

Suppose we have access to a simpler density $Q(\mathbf{x})$ such that:

- We can generate samples from $Q(\mathbf{x})$
- We can evaluate $Q^*(\mathbf{x})$ which is identical to $Q(\mathbf{x})$ up to a constant $Q(\mathbf{x}) = Q^*(\mathbf{x})/Z_Q$

Importance sampling allows us to estimate the expected value of $\phi(\mathbf{x})$ by first generating R samples $\{x^{(r)}\}_{r=1}^{R}$ from $Q(\mathbf{x})$. To correct for the differences between $P(\mathbf{x})$ and $Q(\mathbf{x})$ we introduce a weight for each sample:

$$w_r = \frac{P^*(\mathbf{x})}{Q^*(\mathbf{x})}.$$

The expectation can then be estimated as:

$$\hat{\Phi} = \frac{\sum_{r=1}^{R} w_r \phi(\mathbf{x}^{(r)})}{\sum_{r=1}^{R} w_r}.$$

Importance sampling for Bayesian inference. In the context of Bayesian inference, we have some data $\mathbf{y}$ and we would like to estimate the expected value of a function on the *posterior distribution* of $\mathbf{x}$ given $\mathbf{y}$: $p(\mathbf{x} \mid \mathbf{y})$. Bayes theorem states that:

$$p(\mathbf{x} \mid \mathbf{y}) = \frac{p(\mathbf{x})p(\mathbf{y} \mid \mathbf{x})}{p(\mathbf{y})}.$$

$p(\mathbf{x})$ is called the *prior distribution* of $\mathbf{x}$, and $p(\mathbf{y} \mid \mathbf{x})$ the *likelihood*, or *evidence*.

Consider a situation in which *(i)* we cannot sample directly from the posterior distribution, *(ii)* we can sample from the prior distribution, and *(iii)* given a sample $\mathbf{x}^{(r)}$, we can compute the likelihood $p(\mathbf{y} \mid \mathbf{x}^{(r)})$. Then we can use important sampling to compute the expected value of a function $\phi(\mathbf{x})$ on the posterior distribution by setting:

$$\begin{aligned}
P(\mathbf{x}) &= p(\mathbf{x} \mid \mathbf{y}) \\
P^*(\mathbf{x}) &= p(\mathbf{x})p(\mathbf{y} \mid \mathbf{x}) \\
Q(\mathbf{x}) &= p(\mathbf{x}) \\
w_r &= \frac{P^*(\mathbf{x})}{Q(\mathbf{x})} \\
&= \frac{p(\mathbf{x})p(\mathbf{y} \mid \mathbf{x})}{p(\mathbf{x})} \\
&= p(\mathbf{y} \mid \mathbf{x}).
\end{aligned}$$

That is, we generate samples from the prior distribution and weigh each of them by their likelihood on the data.

Sequential importance sampling. In the context of particle filtering, we have a sequence of observation at time steps 1 to t: $\mathbf{y}_{1:t}$. Each observation $\mathbf{y}_k$ was generated by

Algorithm 4.1 The sequential importance sampling procedure.

```
1: Draw a set of $R$ particles $\{\mathbf{x}_1^{(r)}\}_{r=1}^R$ from $p(\mathbf{x}_1)$
2: Initialise the weights $\{w_1^{(r)}\}_{r=1}^R$ as $w_1^{(r)} = p(\mathbf{y}_1 \mid \mathbf{x}_1^{(r)})$
3: for $i = 2, 3, \dots t$ do
4:     Sample the particles' next time step $\{\mathbf{x}_i^{(r)}\}_{r=1}^R$ from $p(\mathbf{x}_i^{(r)} \mid \mathbf{x}_{i-1}^{(r)})$
5:     Update the weights $\{w_i^{(r)}\}_{r=1}^R$ as $w_i^{(r)} = w_{i-1}^{(r)}\, p(\mathbf{y}_i \mid \mathbf{x}_i^{(r)})$
6: end for
```

a corresponding state $\mathbf{x}_k$. We would like to perform inference over the entire trajectory of states $\mathbf{x}_{1:t}$ given their transition probabilities $p(\mathbf{x}_i \mid \mathbf{x}_{i-1})$. Sequential importance sampling does so by exploiting the recursive nature of the posterior distribution:

$$p(\mathbf{x}_{1:t} \mid \mathbf{y}_{1:t}) = p(\mathbf{x}_t \mid \mathbf{y}_t, \mathbf{x}_{t-1})\, p(\mathbf{x}_{1:t-1} \mid \mathbf{y}_{1:t-1})$$

This exploits the *Markov property*, the distribution state x_t depends only upon x_{t-1} and none of the other states that precede it.

Like importance sampling, sequential importance sampling works by drawing a set of samples, called *particles*, from a distribution and reweighting them appropriately. These weights are computed sequentially as the trajectory of states of the particles are drawn. Algorithm 4.1 shows the procedure. Each particle sequentially samples its trajectory. At each state i, it updates its weight with its likelihood of observing $\mathbf{y}_i$. Like with importance sampling, if we want to compute the expected value of a function $\phi(\mathbf{x})$ at time step i, we can approximate it using the R particles as

$$\hat{\Phi} = \sum_{r=1}^{R} W_i^{(r)} \phi(\mathbf{x}_i^{(r)}) \tag{4.1}$$

where $W_i^{(r)}$ is the normalised weight of particle r at time step i:

$$W_i^{(r)} = \frac{w_i^{(r)}}{\sum_{r'=1}^{R} w_i^{(r')}}.$$

In practice, sequential importance sampling tends to perform poorly. The reason is that with each time step, a fraction of the particles will generate a poor sample from the transition distribution. This results in a low likelihood for those particle. As we go through the time steps, the fraction of particles which will have generated only decent samples throughout their trajectory will decrease exponentially. Eventually, none of the particles will be representative of the distribution.

A side effect of this issue is *degeneracy*: after a few time steps there is a wide distribution of importance weights for the particles, and all but one of the particles have their normalised importance weight $W_i^{(r)}$ near zero. As a result, we are effectively averaging over a single

Algorithm 4.2 The sequential Monte Carlo procedure.

```
 1: Draw a set of $R$ particles $\{\mathbf{x}_1^{(r)}\}_{r=1}^R$ from $p(\mathbf{x}_1)$
 2: Initialise the weights $\{w_1^{(r)}\}_{r=1}^R$ as $w_1^{(r)} = p(\mathbf{y} \mid \mathbf{x}^{(r)})$
 3: for $i = 2, 3, \dots t$ do
 4:     Compute $ESS_{i-1}$
 5:     if $ESS_{i-1} < \frac{R}{2}$ then
 6:         resample particles $\{\mathbf{x}_{i-1}^{(r)}\}_{r=1}^R$ with probability density $\{w_{i-1}^{(r)}\}_{r=1}^R$
 7:         Assign the average weight value $\frac{\sum_{r=1}^R w_{i-1}^{(r)}}{R}$ to each of their weights $w_{i-1}^{(r)}$
 8:     end if
 9:     Sample the particles next time step $\{\mathbf{x}_i^{(r)}\}_{r=1}^R$ from $p(\mathbf{x}_i^{(r)} \mid \mathbf{x}_{i-1}^{(r)})$
10:     Update the weights $\{w_i^{(r)}\}_{r=1}^R$ as $w_i^{(r)} = w_{i-1}^{(r)} p(\mathbf{y}_i \mid \mathbf{x}_i)$
11: end for
```

sample which is unlikely to yield good results. Degeneracy can be quantified by measuring the *effective sample size* (ESS) of the particles at time step i:

$$ESS_i = \frac{1}{\sum_{r=1}^{R} \left(W_i^{(r)}\right)^2}$$

Sequential Monte Carlo. In order to improve over sequential importance sampling, we monitor the ESS throughout the time steps. Whenever it falls below a threshold, typically $R/2$, we perform a *resampling* step which eliminates the less likely particles and duplicates the more likely ones. The full sequential Monte Carlo procedure is shown in Algorithm 4.2. Just like with sequential importance sampling, we can estimate the expected value of a function $\phi(\mathbf{x})$ using Equation 4.1. I describe below the details of resampling.

There are multiple ways in which resampling can be performed. One of the more popular and efficient technique is *systematic resampling.* It samples a single random number $U_1 = \mathcal{U}[0, \frac{1}{R}]$ and defines $U_j = U_1 + \frac{j-1}{R}$. The number of times each particle r is duplicated at time step i is then

$$N_i^{(r)} = \left| \left\{ U_j : \sum_{k=1}^{r-1} W_i^{(r)} < U_j < \sum_{k=1}^{r} W_i^{(r)} \right\} \right| .$$

If $N_i^{(r)}$ is 0, particle r is deleted. The values $U_1 \dots U_R$ can be viewed as regular "ticks" in the interval [0,1]. To compute $N_i^{(r)}$ we allocate each of the particles a region of size $W_r^{(i)}$ in the range [0,1] and count how many ticks fall within each particle's region. After a resampling step, all particles can be seen as drawn uniformly from the posterior distribution. We therefore re-initialise their weights equally. Typically the value $\frac{1}{R}$ is used, but I use instead the average values of the weights $\sum_{r=1}^R w_i^{(r)}/R$ for reasons I explain later.

The limits of SMC In order to produce distributions similar to the true posterior, SMC relies on our ability to sample from the transition $p(\mathbf{x}_i \mid \mathbf{x}_{i-1})$ and get some particles which followed a similar behaviour as the one which generated the data. In low dimensional state space models, such as when tracking a physical object in a 3 dimensional space, this is a reasonable assumption. However, we must be aware that, due to the curse of dimensionality, SMC will perform poorly if a transition samples random values in a large multi-dimensional space. An important part of Probabilistic-C++, presented in the next section, will be to allow users to expose some independence in the distributions being modelled to tackle these large dimensional spaces.

Another limitation comes from the resampling step. Because some of the less likely particles are discarded each time we resample, as more time steps are performed, fewer of the original particles will remain. This is another aspect of degeneracy. Most of the original particles have probability 0 assigned to them as they are discarded in later time steps. This pitfall is the reason we only resample when the effective sample size has decreased to low values and not every iteration. However, we should still be aware that too many resampling steps will eventually lead to degeneracy.

Likelihood of the model A nice property of SMC is that it not only allows us to perform inference over a model, but also allows us to quantify how good that model was at explaining the data. In the context of Bayesian inference, this is measured by the likelihood of the model. Following the SMC procedure, we can estimate the likelihood of the model at time step i by measuring the average weight of the particles $\sum_{r=1}^{R} w_i^{(r)}/R$. This is why I did not reinitialise them to $\frac{1}{R}$ in my description above.

Applying SMC to probabilistic programs SMC is directly applicable to probabilistic programs. Random draws are seen as transition probabilities. Observe statements are the observations $\mathbf{y}$. The only necessary construct is the ability to duplicate a program's execution thread into two identical threads so we can perform the resampling step. A good analogy is the POSIX `fork()` construct which duplicates a single threaded process. This is in fact used by the probabilistic programming language Probabilistic-C [PW14] to perform inference. I describe this mechanism in more details below.

To perform inference on a probabilistic program, we initiate R execution threads executing the program concurrently. Each thread will correspond to a particle and keep track of its own weight $w^{(r)}$. In order to sample from the transition distribution, we simply execute the probabilistic program, sampling random values whenever necessary. We execute all threads up to the first `observe` statement, at which point they halt in a synchronous barrier. The threads use the content of the `observe` to update their weights $w^{(r)}$ accordingly.

A master thread then gathers each of the threads weights and checks whether the effective sample size is below a threshold. If it is, the master thread samples which of the threads should be duplicated, and which should be terminated. It then instructs all threads to act accordingly. We repeat this procedure of synchronous barrier and possible resampling for each `observe` statement in the program. At `predict(expr)` statements, threads output their computed value for `expr`. We can then compute the average value of `expr` by performing a weighted average, as in Equation 4.1.

Finally, it is important to mention that there have been recent breakthroughs in research on sequential Monte Carlo, with the recent introduction of particle Markov chain Monte Carlo methods (PMCMC) [ADH10]. These hybrid techniques combine the SMC and Markov chain Monte Carlo (MCMC) inference algorithms by using SMC as the inner loop of an MCMC. PMCMC techniques have been applied to probabilistic program inference. Unfortunately, for an identical computational budget, they tend to perform slightly worse than SMC [PW14].

4.2 Probabilistic-C++

This section presents Probabilistic-C++, BOAT's lightweight and high performance probabilistic programming library. It is implemented in C++ and based on the recently proposed Probabilistic-C programming language. I first introduce the interface offered by Probabilistic-C++ through an example program (Section 4.2.1) and discuss its relationship with existing frameworks (Section 4.2.2). Section 4.2.3 lists some of Probabilistic-C++'s useful features. Finally, I show how model independence can be exploited to help the inference converge (Section 4.2.4).

4.2.1 Interface

In this subsection, I review how models are implemented in Probabilistic-C++. I use linear regression as a running example, which could be useful, for example, to model the runtime of an $\mathcal{O}(n)$ algorithm.

To declare a probabilistic model, developers declare a *model class* in C++. The Bayesian implementation of one dimensional linear regression has three parameters: `alpha` the slope, `beta` the y-intercept, and `noise` the amount of noise at each data point. Listing 4.4 shows the Linear regression model class with those three parameters as fields. On a high level, the role of inference will be to find *model objects* instances of the model class whose parameters fit the data well.

A model class must implement three functions. First, the constructor which samples parameter values from the *prior* distribution. The prior distribution represents the uncertainty on the parameter values before any data points have been observed. In the example, `alpha` and `beta` are drawn from normal distributions with mean 0.0 and standard deviation 1.0, and `noise` is assigned 2.0. Second, the `observe` function which will

```cpp
struct LinearModel {
  LinearModel() {
    alpha = std::normal_distribution<>(0.0, 1.0)(generator);
    beta  = std::normal_distribution<>(0.0, 1.0)(generator);
    noise = 2.0;
  }

  double observe(double x, double y) {
    double prediction = alpha * x + beta;
    return normal_lnp(prediction, y, noise);
  }

  double predict(double x) {
    return alpha * x + beta;
  }

  double alpha, beta, noise;
};

int main() {
  ProbEngine<LinearModel> engine;
  engine.observe(1.0, 3.0);
  // Average value predicted at x=0.0 (beta)
  std::cout << engine.predict(0.0) << std::endl;
}
```

Listing 4.4: Linear regression in Probabilistic-C++.

be used to perform inference. An `observe` takes as argument a data point, and returns the object's log-probability of producing that point. In the example, this is done by comparing `prediction`, the value predicted by the object's parameters, with the observed value `y`. We use the `normal_lnp` function – part of Probabilistic-C++ – which returns the log-probability of getting `y` from a normal distribution draw centred on `prediction` and with standard deviation `noise`. Third, the `predict` function allows the model to be queried. Given a value `x`, it returns the value predicted at `x`.

To use a model class, users declare a probabilistic engine `ProbEngine` templated on that class. On construction, the engine will create many model objects, always using the default constructor to do so. These will correspond to particles when performing SMC inference. Developers can then make the engine `observe` data points. The engine will call the corresponding `observe` function on each of its model objects and use the returned log-probabilities to determine which of the objects best fit the data. In the example, I make the engine observe the data point $x = 1.0, y = 3.0$. Intuitively, this means objects with higher values of `alpha` and `beta` will be considered more likely. Finally, the engine can be queried using the `predict` function which will return the average of the predictions of the model objects. In the example, I predict the average value of `beta`, which is near 0.75.

Framework	Runtime	Peak memory usage
Probabilistic-C	52.090s	15840 MB
Probabilistic-C++	0.035s	6 MB

Table 4.1: Measured performance on the linear regression example with 100,000 particles. Ran on an r3.2xlarge EC2 instance with 8 hyperthreads and 61 GB of memory.

4.2.2 Relationship with existing probabilistic programming frameworks

Probabilistic-C++'s architecture is based on that of Probabilistic-C [PW14] – both use Sequential Monte Carlo (SMC) [DDFG01] to perform inference. In Probabilistic-C++, SMC duplicates likely model objects, using the C++ assignment operator, and discards others to keep a pool of particles with balanced likelihoods. The key difference between the two frameworks is that in Probabilistic-C particles are processes and duplication is done using POSIX `fork()`. Using `fork()` allows Probabilistic-C to be a probabilistic programming *language*, where the entire program can be viewed as a distribution and each process is a particle. On the other hand, it suffers from the overhead of `fork()`, which is orders of magnitude slower than the C++ assignment operator. For illustration, I compare the computational performance of the two frameworks on the linear regression example in Table 4.1. Probabilistic-C++ reduces the runtime and memory consumption of inference by over 1000×. Note that Probabilistic-C++ makes no algorithmic improvement over Probabilistic-C, and I therefore do not evaluate its regression performance, referring the reader to [PW14].

Compared to a traditional probabilistic languages such as Venture [MSP14] or Anglican [TvdMW15], we retain the ability of programming complex distributions without needing an entire new language. Probabilistic-C++ is integrated within BOAT. The models implemented in Probabilistic-C++ are Turing-complete, they can execute any C++ function. Some probabilistic programming languages such as Stan [CGH$^+$16] restrict the class of allowed models to employ more advanced inference algorithms. In practice, I find Turing-completeness to be useful when modelling complex computer systems.

4.2.3 Features and limitations of Probabilistic-C++

I now go over some of the features available in Probabilistic-C++ that I have found to be useful and are not always available in probabilistic programming languages. The subsection concludes on some of the limitations of Probabilistic-C++.

Observes are top level. In many but not all probabilistic programming languages, `observe` calls are forced to be top level. This means it is not allowed for an `observe` statement to be executed conditionally on the value of another variable. This is because the resulting distribution would be ill formed. This is not enforced in Probabilistic-C. For this reason, Probabilistic-C calls itself a "compilation target for probabilistic programs" rather than a probabilistic language. It is designed to be a fast backend to higher level probabilistic languages. Probabilistic-C++ forces `observe` calls to be top level as they are called on the `ProbEngine` rather than within the probabilistic code itself. This guarantees that the content of the observe will be executed on all particles. This makes it suitable to be a probabilistic programming library, which can be exposed to users.

Decoupled implementation of the model from the data used for inference. Some probabilistic programming frameworks mix definitions of probabilistic models with the data used for inference. This is, for example, the case in the Python-like probabilistic program pseudo-code from section 4.1.1. In Probabilistic-C++ the two are decoupled, making the model definitions simpler.

Decoupled predictions from the model. Another decoupling is that we can separate the model definition from the statistics we are trying to gather about the model. In the linear regression example I predicted the mean value of `beta` in the posterior distribution. If we want to average more complex expressions, we can do so using C++ lambda functions. Say we had computed the mean value of `beta` as `beta_mean`, the variance could be computed as:

```
beta_variance = engine.average(
  [&](const LinearModel& m){
    return pow(m.beta - beta_mean, 2.0);
  });
```

Incremental observes. Most probabilistic programming languages are designed to perform inference on a static dataset. As described above, the data may even be included in the program. Within the context of Bayesian optimisation, each iteration leads to new data that must be incorporated in the model. Therefore, when receiving the kth observation, it is beneficial to be able to re-use the inference performed for the $k-1$ previous samples rather than starting from scratch. In Probabilistic-C++, this is the default behaviour; `observe` calls are performed incrementally, internally performing another time step of SMC. In contrast, in Probabilistic-C, because `observe` statements are part of the compiled program, the inference would have to be run from scratch.

Deterministic observes. The resampling step used by SMC allows likely particles to be replicated many times. In turns, this means when an `observe` call performs random actions, the likely particles will get many more "shots" at choosing the action that best explains the data. The downside of resampling is that it eventually leads to degeneracy, as explained in Section 4.1.2. When the `observe` call performs no random actions, we would hence like to avoid this resampling. `ProbEngines` in Probabilistic-C++ includes an `observe_deterministic` method, allowing them to perform an `observe` without the corresponding resampling step.

```cpp
template <class Archive>
void LinearModel::serialize(Archive& ar, const unsigned int version) {
  ar & alpha;
  ar & beta;
  ar & noise;
}

int main() {
  ProbEngine<LinearModel> engine;
  engine.observe(1.0, 3.0);
  engine.store("linear_model.txt");
}
```

Listing 4.5: Serialisation example in Probabilistic-C++.

Quantifying the quality of a model. Performing inference on a model is useful. However, sometimes we would like to query the quality of the model itself. In Bayesian statistics, this can be measured through the *marginal likelihood*. In Probabilistic-C++, given a `ProbEngine engine`, we can query the logarithm of the likelihood as `engine.get_marginal_log_likelihood()`. As described in Section 4.1.2, this can be readily computed when using SMC for inference by averaging the likelihoods of all particles.

Dynamically changing the number of particles. Having many particles allows the inference to be more accurate, but it also leads to a greater computational cost. Sometimes we would like to adjust the number of particles based on the difficulty of performing inference at a given time step. Given a `ProbEngine`, this can be done in Probabilistic-C++ by calling `engine.set_num_particles(int)`. The engine will perform an extra resampling step to adapt the number of particles. Typically, this will be used to have very large numbers of particles for the first few observations, so we can have many samples from the prior distribution. Afterwards, once the randomness only comes from the `observe` calls, we can reduce the number of particles for the rest of the computation and pay a lower computational cost.

Serialising probability distributions. It is often useful to be able to save an inferred distribution to a file so it can later be reused, either for prediction or further inference. In Probabilistic-C++, this can easily be achieved by implementing a `serialize` member function of model classes using Boosts serialisation library [Boo16]. This can be used to store all properties of a probabilistic engine, including all of its particles, to a file so it can be loaded again later. Listing 4.5 shows how this is implemented in the `LinearModel` example by adding a member function.

I now discuss some of the limitations of Probabilistic-C++ and its underlying SMC inference algorithm.

SMC as the only inference algorithm. In Probabilistic-C++, SMC is the only available inference algorithm. In contrast, many probabilistic programming languages offer access to a variety of inference methods such as Metropolis-Hastings, Gibbs sam-

```
int sort_and_hash(vector<int>& array){
  sort(array.begin(), array.end());
  int result = hash(array);
  return result;
}
```

Listing 4.6: Implementation of `sort_and_hash`.

pling, slice sampling, Hamiltonian Monte Carlo or variational inference [TKD+16, RG15, MSP14, TvdMW15]. Different inference algorithms will be best suited to different types of models. Hence allowing a variety of inference methods increases the range of probabilistic programs that can be performed inference over. By only allowing the use of SMC, Probabilistic-C++ can therefore only perform inference over a narrow range of models. I now discuss which models fall within that range.

Limitations of SMC. Typically, most of the random draws done by a model in Probabilistic-C++ will be performed in the constructor, when we sample the model parameters. As described in Section 4.1.2, SMC will perform poorly when we perform too many draws at once with no observation in-between informing us of the quality of those draws. Hence, if the model's prior distribution samples too many parameters at once, inference will perform poorly. Typically we will be using on the order of 10^5 particles. In practice, this means we should not expect to successfully perform inference on more than five continuous model parameters accurately. This is a clear limitation and if it is not tackled, we will not be able to model complex systems accurately. The next subsection describes how a user can expose the *independence* and *conditional independence* available in a model to help the inference converge.

4.2.4 Exploiting model independence

This subsection discusses the exposition of model independence in Probabilistic-C++. As a running example, I will attempt to model the execution time of three functions: `sort()` which sorts an array, `hash()` which hashes an array, and `sort_and_hash` which sorts and then hashes an array. I assume that executing `sort_and_hash` requires the same amount of time as executing `sort()` and `hash()` subsequently. Listing 4.6 shows its implementation. Figure 4.1 shows the graphical representation of the model and the data. I assume we already have the model classes `SortModel` and `HashModel` which model the execution time of `sort()` and `hash()` as a function of their parameters.

I first show the way this model would traditionally be implemented in probabilistic programming (Section 4.2.4.1). This implementation may require too much inference work to be performed by the model and could lead to poor inferred parameters. I then present a way to leverage the independence of the model to help the model converge, and show its implementation in Probabilistic-C++ (Section 4.2.4.2). This implementation assumes we only have data from `sort()` and `hash()` and none from `sort_and_hash`, but we still

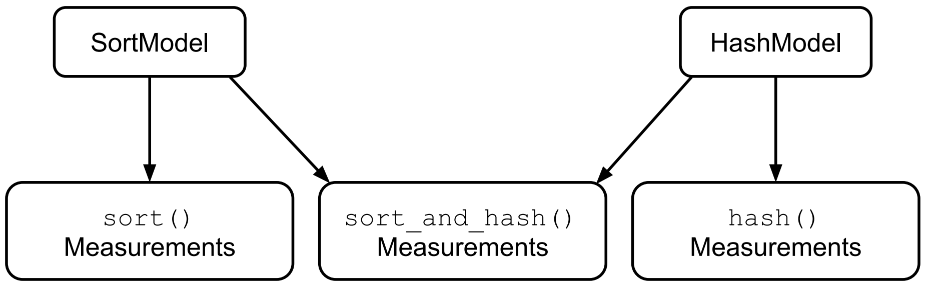


Figure 4.1: Graphical representation of the `sort()` and `hash()` models and the data.

would like to be able to make predictions for `sort_and_hash`. I show how this method also applies to conditional independence using the garbage collection case study model as an example (Section 4.2.4.3). Finally, I show we can leverage both independence and measurements from `sort_and_hash`, albeit at the cost of incremental observes (Section 4.2.4.4).

4.2.4.1 Traditional Probabilistic Programming implementation

The traditional way to express the graphical model of Figure 4.1 in probabilistic programming is as follows. The entire model would be placed in a single model class. Listing 4.7 does this by placing a `SortModel` and `HashModel` into a combined class `FullModel`. The default constructor of `FullModel`, implicitly defined with C++'s semantics, will call the default constructors of `SortModel` and `HashModel`. The issue with this approach is that the parameters of both `SortModel` and `HashModel` will be sampled at once in `FullModel`'s constructor. Hence, the total number of model parameters being sampled by the constructor of `FullModel` may be high. This may lead to few model objects having parameters which explain the data well.

4.2.4.2 Exposing model independence

I now show how some structure of the probabilistic model can be leveraged to help the inference engine. There is a notion of independence that can be extracted. `SortModel` and `HashModel` both are sampled independently and perform inference on different data (assuming we have no data for `sort_and_hash`). But we still would like to perform predictions based on their joined distribution. Probabilistic-C++ offers a way to do this. In Listing 4.8, I implement two separate `ProbEngine`s for the two models. I then use the Probabilistic-C++'s `average` function to average a prediction over multiple engines.

The first time the function `value()` is called on an engine within an `average`, the engine performs a temporary resampling steps. This is so that each of its model objects have identical probabilities, and it has as many model objects as the number of samples used for `average` (a parameter passed implicitly). Then, if `value()` is called at the ith iteration

```
class FullModel {
  // Three types of predictions
  double predict(const SortParameters& parameters) {
    return sort_model.predict(parameters);
  }

  double predict(const HashParameters& parameters) {
    return hash_model.predict(parameters);
  }

  double predict(const SortParameters& sort_parameters,
                 const HashParameters& hash_parameters) {
    // Combine both models' predictions
    return sort_model.predict(sort_parameters) +
           hash_model.predict(hash_parameters);
  }

  // Three types of observes
  double observe(const SortParameters& parameters, double runtime){
    return sort_model.observe(parameters, runtime);
  }

  double observe(const HashParameters& parameters, double runtime){
    return hash_model.observe(parameters, runtime);
  }

  double observe(const SortParameters& sort_parameters,
                 const HashParameters& hash_parameters, double runtime){
    // Compare the combined model predictions with the data
    double prediction = sort_model.predict(sort_parameters) +
                        hash_model.predict(hash_parameters);
    // Assumes a noise of 0.1s
    return normal_lnp(prediction, runtime, 0.1);
  }

  SortModel sort_model;
  HashModel hash_model;
};

int main(){
  ProbEngine<FullModel> engine;
  // Use the model for inference on any type of data
}
```

Listing 4.7: Implementation of a model combining SortModel and HashModel.

```
// Assume we have some data in sort_measurements and hash_measurements
// And that we want to predict the runtime of sort_and_hash in sah_parameters
ProbEngine<SortModel> sort_model;
ProbEngine<HashModel> hash_model;

// Perform inference on the models independently
for(const auto& m : sort_measurements){
  sort_model.observe(m.sort_parameters, m.runtime);
}
for(const auto& m : hash_measurements){
  hash_model.observe(m.hash_parameters, m.runtime);
}

// Use both models to make a prediction
double prediction = average(
  [&](){
    // Use value() to retrieve an engine's content
    return sort_model.value().predict(sah_parameters.sort_parameters) +
           hash_model.value().predict(sah_parameters.hash_parameters);
  });
```

Listing 4.8: Performing observations on `SortModel` and `HashModel` independently and using their joint distribution for predictions.

of `average`, the engine returns the ith of its model objects. Once `average` returns, the resampling is undone to avoid causing degeneracy. An arbitrary number of engines can be used in a function being averaged.

Note that in the context of Listing 4.8, where we predict the average `sort_and_hash` time, this approach is not necessary as we could simply add the predictions of each model independently. However, the approach is also applicable to models whose predictions are not simply summed together.

4.2.4.3 Exposing model conditional independence

In a similar way to how we exploited independence above, we can exploit *conditional independence*. Consider the garbage collection model presented in Section 3.2.1. The model is composed of three parts, first `GCRateModel` and `GCDurationModel` predict the rate and average duration of the garbage collections respectively as a function of the GC flags values. Then `LatencyModel` predicts the 99th percentile latency of the database as a function of the rate and duration of GCs and the flag values. Whenever we perform an experiment and measure empirical data, we collect the true rate and duration of GCs as well as the 99th percentile latency.

Note that given the true rate and duration of GCs, `LatencyModel` is independent of both `GCRateModel` and `GCDurationModel`. This is called conditional independence, a property

```cpp
ProbEngine<GCRateModel> rate_model;
ProbEngine<GCDurationModel> duration_model;
ProbEngine<LatencyModel> latency_model;

// Perform inference on the models independently
rate_model.observe(gc_flags, result.rate);
duration_model.observe(gc_flags, result.duration);
latency_model.observe(gc_flags, result.rate,
                      result.duration, result.latency);

// Predict the latency of gc_flags
double latency_prediction = average(
  [&](){
    double rate = rate_model.value().predict(gc_flags);
    double duration = duration_model.value().predict(gc_flags);
    return latency_model.value().predict(gc_flags, rate, duration);
  });
```

Listing 4.9: The overall model of the garbage collection case study, originally described as a graph in Figure 3.2.

often used in graphical models, and key to some inference algorithms such as Gibbs sampling [Mur12]. To exploit it here, we simply perform inference on each of the models separately. When predicting a 99th percentile latency, we can use the `average` function as we did above. Listing 4.9 shows the high level implementation of the garbage collection model using `average`.

4.2.4.4 Combining models for inference

Finally, I go back to the `sort_and_hash` example and consider the case where we also have measurements from `sort_and_hash`. In this context, we must use the `FullModel` data structure to perform inference on those measurements. However, we can still use the data from `sort` and `hash` to perform inference on their specific part of the model initially. Listing 4.10 shows how to do this. At the top, we implement a new constructor for `FullModel` which takes as input its `SortModel` and `HashModel` parameters. This will allow us to create `FullModel` objects which come from a specific distribution, different from the default constructor's one.

Underneath, we create two `ProbEngines` for `SortModel` and `HashModel` to perform inference on their respective data. We then construct a third `ProbEngine` for `FullModel`. We use Probabilistic-C++'s method `create_from_engines` to construct its particles from the other two engines. Internally, this combines the particles of both `ProbEngines` and uses `FullModel`'s new constructor to combine them into `FullModel` particles. Once this is done, we can use the `sort_and_hash` measurements to perform inference on the `FullModel` `ProbEngine`.

When compared to the traditional probabilistic programming approach, as presented

```
// Implement a new constructor for FullModel
FullModel::FullModel(SortModel sm, HashModel hm)
  : sort_model(sm),
    hash_model(hm){
}
/*-----------------------------------------------------------------------*/
// Perform inference using all types of measurements
ProbEngine<SortModel> sort_model;
ProbEngine<HashModel> hash_model;

// Exploit the independence fot the sort() and hash() measurements
for(const auto& m : sort_measurements){
  sort_model.observe(m.sort_parameters, m.runtime);
}
for(const auto& m : hash_measurements){
  hash_model.observe(m.hash_parameters, m.runtime);
}

// Combine both models into one
ProbEngine<FullModel> full_model;
full_model.create_from_engines(sort_model, hash_model);

// Use the combined models to perform inference
for(const auto& m : sort_and_hash_measurements){
  sort_model.observe(m.sort_parameters, m.hash_parameters, m.runtime);
}
```

Listing 4.10: Combining independent models to perform joint inference.

in Section 4.2.4.1, the implementation presented here has the following advantages and disadvantages.

Advantage : Better convergence. Given a dataset and a fixed amount of computational power, the implementation presented here will converge better to the true posterior distribution.

Disadvantage : More complex implementation. This makes the implementation more error prone. Users should only implement it if they do require the better convergence brought by the technique.

Disadvantage : No $\mathcal{O}(1)$ incremental observation. This is because the order in which observations now matters, we must first observe all measurements associated with individual models before they are combined. Hence, unlike Section 4.2.4.1's implementation, the implementation here cannot perform a small amount of incremental work per new observation. For example, say new data of `sort`'s performance is made available. We can integrate this data incrementally into `sort_model`. But `sort_and_hash_model` will have to be constructed from scratch with the newly inferred `sort_model` particles. All of the `sort_and_hash` measurements will have to be used for inference again. Hence, the cost of adding a new measurement in the model will go from $\mathcal{O}(1)$ to $\mathcal{O}(n)$, where n is the amount of data available for inference.

In practice, I never use this last approach. Instead, I design my models so that each model class is used to predict one specific type of measurement, and use the true value of these measurements to exploit conditional independence. The next section discusses the construction of realistic models using a semi-parametric approach.

4.3 Probabilistic Programming in Practice

Developers build bespoke auto-tuners in BOAT by declaring a probabilistic model of the system's behaviour. This section first presents *semi-parametric* models [Mur12], an easy to use class of probabilistic models that are well suited to SBO (Section 4.3.1). I then show how a semi-parametric model can be used in Probabilistic-C++ via an example from the garbage collection case study (Section 4.3.2). Finally, I list some design considerations for models used in SBO (Section 4.3.3).

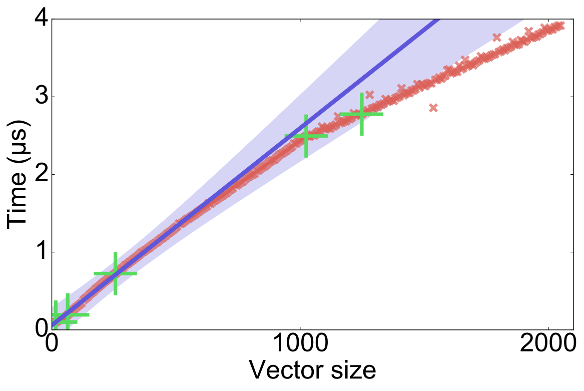
4.3.1 Semi-parametric models

There are two desirable properties that a model should have in the context of SBO:

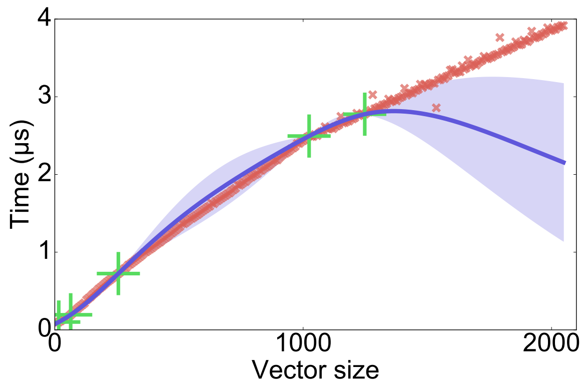
- It should understand the general trend of the objective function to avoid exploring regions of low performance. This wastes iterations and, when optimising runtime, can lead to longer evaluations.
- It should have high precision in the region of the optimum, to find the point with highest performance.

Semi-parametric models, which I now describe, can fulfil both properties. They are a combination of *parametric* models and *non-parametric* models. As a running example, I model the average time needed to insert an element into a sorted vector as a function of its length. This has complexity $\mathcal{O}(n)$, but implementations will have runtimes affected by cache effects and other hardware properties. Figure 4.2 compares the predictions of a parametric, non-parametric and semi-parametric model after observing five points from the dataset. The data was obtained using the `boost::flat_set` data structure [Boo16] and averaged over a million runs.

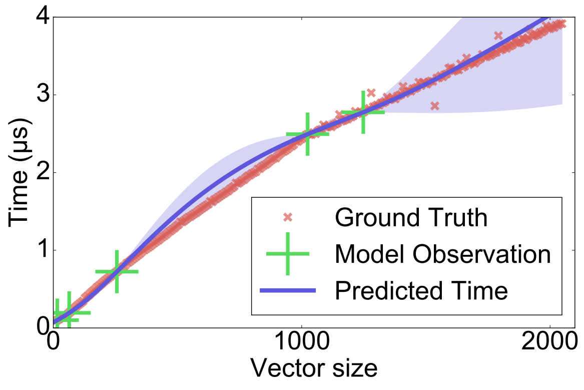
Parametric models learn a fixed number of parameters. For example, `LinearModel`, from Section 4.2.1, is a parametric model that learns two parameters `alpha` and `beta`. Parametric models allows developers to specify the expected behaviour of the system. In the



(a) Parametric (Linear regression)



(b) Non-parametric (Gaussian process)



(c) Semi-parametric (Combination)

Figure 4.2: Three models predicting the time to insert an element into a sorted vector after five observations.

example, this means specifying that the relationship between length and time is linear and not, for example, quadratic. They, however, cannot fit subtleties in the data. I fit a linear regression to five data points from the sorted-vector data in Figure 4.2a. Although the general trend is correct, the model fails to fit all of the data points as they are not strictly linear.

On the other hand, *non-parametric* models learn an unbounded number of parameters that grows with the training data. For example, in the k-nearest neighbour algorithm each training example is memorised so it can be used for prediction. Non-parametric models typically provide no direct way to specify a general trend. Gaussian Processes (GPs), used in traditional Bayesian optimisation, are a powerful family of non-parametric models. I use a GP to fit the same five points from the sorted-vector data in Figure 4.2b. They succeed at fitting all of the data points, but fail to grasp the overall trend. In the context of Bayesian optimisation, this can lead to the over exploration of regions with poor performance. This may seem acceptable for the sorted vector example, but the number of these regions grows exponentially with the number of dimensions.

Semi-parametric models combine a parametric model and a non-parametric one. The non-parametric model is used to learn the difference between the parametric model and

```
class GCRateModel{
  GCRateModel(){
    // Prior distribution on the parameters
    allocated_mbs_per_sec = uniform_draw(0.0, 5000.0);
    gp.stdev(uniform_draw(3.0,30.0));
    gp.linear_scales({uniform_draw(0.0, 15000.0),
                      uniform_draw(0.0, 20.0)});
    gp.noise(uniform_draw(0.001, 0.01));
  }

  double parametric(int ygs, int sr){
    //Compute the size of eden used by the JVM
    double eden_size = ygs * sr / (sr + 2);
    return allocated_mbs_per_sec / eden_size;
  }

  double predict(int ygs, int sr, int mtt) {
    return gp.predict({ygs, sr, mtt}) +  parametric(ygs, sr);
  }

  double observe(int ygs, int sr, int mtt, double observed_rate){
    return gp.observe({ygs, sr, mtt},
                      observed_rate - parametric(ygs,  sr));
  }

  double allocated_mbs_per_sec;
  GaussianProcess gp;
}
```

Listing 4.11: Semi-parametric model of the rate of garbage collections.

the observed data. In Figure 4.2c, I fit the sorted-vector data with a semi-parametric model that simply combines the previous two models. Predictions interpolate all data points, and correctly keep increasing with larger vector sizes.

It is interesting to note that some of the earlier work on Gaussian Processes was concerned with performing inference over such semi-parametric models. For instance, O'Hagan [O'H78] presents a method to perform full Bayesian inference on semi-parametric models where the parametric part of the model is of the form $\mathbf{h}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\beta}$, $\mathbf{h}()$ is a fixed set of basis functions and a Gaussian prior is placed on the parameters $\boldsymbol{\beta}$.

4.3.2 Semi-parametric models in Probabilistic-C++

To build a semi-parametric model in Probabilistic-C++, both the parametric and non-parametric parts of the model must be included in the model class. The non-parametric model is used to model the difference between the parametric model and the observed data.

Probabilistic-C++ comes with its own Gaussian Process implementation which can be used as for non-parametric models. It is implemented on top of the high performance Eigen library for linear algebra [GJ+10]. It implements the incremental version of Cholesky decomposition at an $\mathcal{O}(n^2)$ complexity, as described in Section 2.1.2. Its API also makes its intuitive to use in the context of Probabilistic-C++.

For example, recall from Section 3.2.1 that in the context of the garbage collection (GC) case study, the rate of GCs is inversely proportional to the size of the eden heap region of the JVM. The `GCRateModel` class, shown in Listing 4.11, is a semi-parametric model. It predicts the rate of GCs as a function of the young generation size flag (`ygs`), the survivor ratio flag (`sr`), and the max tenuring threshold flag (`mtt`).

It has two fields: `allocated_mbs_per_sec`, a probabilistic variable part of the parametric model, and `gp`, a Gaussian process as the non-parametric model. The constructor assigns values from the prior distribution to `allocated_mbs_per_sec` and `gp`'s hyperparameters. The parametric model computes the size of the eden region as a function of the flag values and returns its estimate for the rate of garbage collections. In `predict` and `observe`, the Gaussian process object `gp` models the difference between the parametric model and the true rate as a function of the flag values.

Although simple, this model understands the general behaviour of the rate of GCs as a function of the flag value.

4.3.3 Designing models for bespoke auto-tuners

In conclusion, here are the main takeaways of how a probabilistic model should be designed in the context of an SBO:

- **Create one model class per type of collected runtime measurements if possible.** Runtime measurements will often allow to exploit conditional independence, as show in Section 4.2.4. This helps the inference algorithm converge. Hence, it is best to build the model classes as a function of the measurements available.
- **Make each model class semi-parametric.** For reasons presented above, semi-parametric classes will both be able to interpolate the data and generalise well.
- **Only add structure to the parametric part of models if it has shown to converge too slowly.** In the context of SBO, non-parametric models require little developer effort. Including parametric parts to describe general behaviour will help the optimisation converge faster but requires some analysis. In my experience, complex parametric models designed in a narrow setting may also fail to generalise. Therefore, when building a model for an SBO, I recommend beginning with non-parametric models. Then, add structure to the model until the optimisation converges in a satisfactory time.

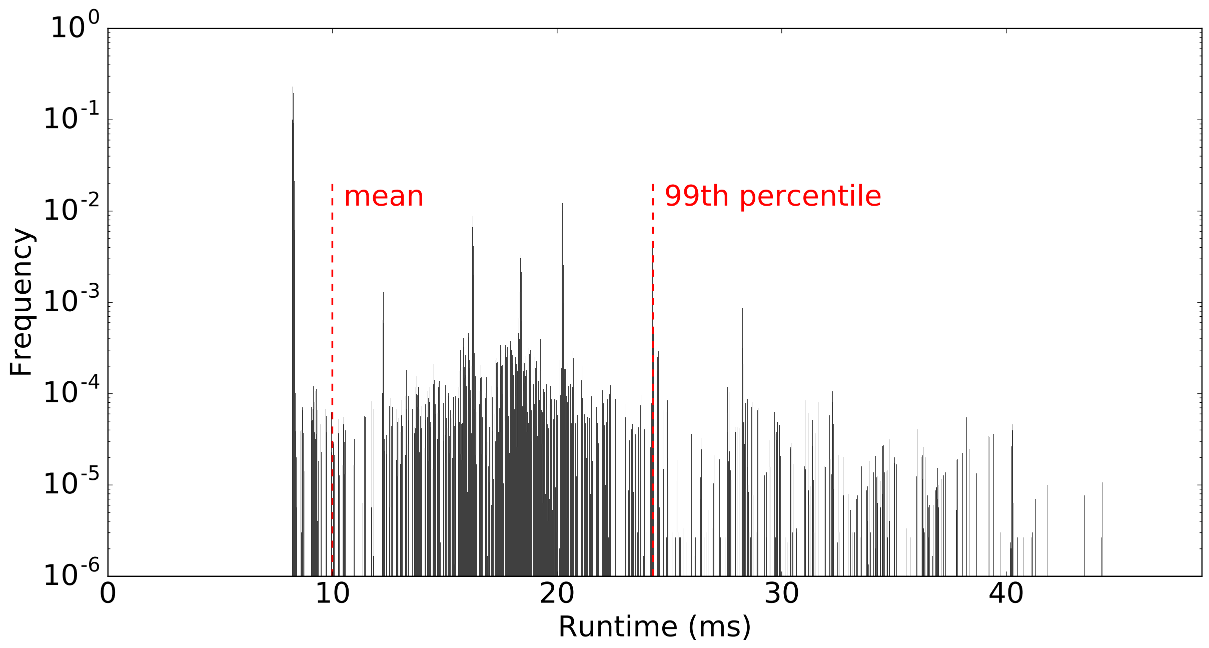


Figure 4.3: Distribution of runtimes of a fixed computation in a shared resource environment.

- **Limit the parametric parts of models to five parameters or fewer.** The SMC algorithm used for inference will not be able to tackle inference over large multi-dimensional prior distributions. Hence, we should limit the number of parameters sampled in the prior distribution or the inferred parameter values will be poor.

4.4 Inference with non-Gaussian likelihoods

A frequent difficulty when benchmarking short lived computer programs is the high variance of their runtime. Measuring the execution time of the same program multiple times will often show a complex long tailed distribution. This is especially true if there are other concurrent programs being executed on the same machine or system, and they are competing for resources.

To illustrate this, I repeatedly perform the same computation, consisting of simple local floating points operations, on an `m4.xlarge` EC2 instance with four hyper-threads. Concurrently, I execute four infinite bash loops designed to use CPU. Hence, there are a total of five running threads competing for the machine's four hyper-threads. Figure 4.3 show the distributions of observed runtimes of the simple floating point operation. We see that although there is a clear mode slightly below 10ms, the distribution has a complex multi-modal long tail.

This behaviour makes inference difficult. I find that standard long-tail statistics distributions, such as log-normal distributions, tend to underestimate the length and thickness of the tail. If we are doing inference with a semi-parametric model using a Gaussian process, as suggested in the previous section, they will use a Gaussian likelihood to model noise which is particularly unsuited to this noise distribution. When observed, the outliers will produce low likelihoods, unless we use a high noise standard deviation parameter with the Gaussian process in which case it will learn slowly.

The traditional approach to tackle this problem is to repeat the execution many times and average the results. The central limit theorem guarantees that as long as the executions are independent, the distribution of these averages will be normally distributed around the true mean. However, this approach is clearly wasteful of resources. Some information is contained in the short measurements and we should consider how it can be exploited. In the context of SBO, we will rarely optimise the execution time of short programs, but we could be optimising the execution time of a long program which generates many short measurements.

In this section, I consider the task of modelling the long tailed noise distribution of computations. I present three contributions:

- I discuss a generic method to perform Gaussian process inference on measurements with an input-dependent non-Gaussian noise distribution using particle filtering algorithms and pseudo-measurements (Section 4.4.1). Although the general approach is not new, I am not aware of prior work applying it to the context of Gaussian processes.
- I present a way to empirically generate an approximate noise distribution for the execution of single threaded computer programs, and to use this distribution for inference (Section 4.4.2).
- I evaluate this method on a matrix multiplication dataset. The goal is to infer the average runtime of matrix multiplications based on the dimensions of the matrices (Section 4.4.3).

4.4.1 Gaussian process inference on non-Gaussian noise functions with particle filters

Consider performing inference on a Gaussian process within a particle filter as presented in the previous section. We are modelling a function $f()$ such that

$$f(\mathbf{x}) \sim \mathcal{GP}(0, k(\mathbf{x}, \mathbf{x}')).$$

In a noisy environment, we are trying to perform inference from a measurement y:

$$y = f(\mathbf{x}) + \varepsilon$$

where ε is distributed from a noise function $\varepsilon \sim p_n(\varepsilon \mid \mathbf{x})$. I make the assumption that $p_n()$ is only dependent of the value of $f()$ at that point: $p_n(\mid \mathbf{x}) = p_n(\mid f(\mathbf{x}))$. Further, I make the assumption that $p_n()$ can only be sampled from. We do not have access to its likelihood.

If $p_n()$ is not a normal distribution, we can not feed a measurement y directly into the Gaussian process. Instead, we have to create a pseudo-measurement y'. I describe two ways of doing this. First, the traditional probabilistic programming approach, based on forward sampling. Second, an approach that uses backward sampling. It assumes that from a measurement y, we can sample backward from the noise distribution $p_n(f(\mathbf{x}) \mid y)$.

In the first approach, we sample a pseudo-measurement y' from $\mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))$ and make the GP observe it. In Probabilistic-C++, this can be done in a single step using `GaussianProcess::sample_and_observe()`. Then, we sample a noise ε from the distribution $p_n(\mid y')$.

At that stage, we have the model's prediction for the measurement $y' + \varepsilon$ and the true measurement y. If we were rigorous, we would return as the particle's likelihood 1 if $y = y' + \varepsilon$ and 0 otherwise. Since we are modelling continuous values, its unlikely that both values will ever match. Instead, we allow for some small error from the sampling an use a normal likelihood centred on $y' + \varepsilon$ with variance σ_n^2. In Probabilistic-C++, this means returning `normal_lnp(`$y' + \varepsilon$`, `y`, `σ_n`)`.

There are two issues with this approach. First, we do not use the GP's ability to model Gaussian noise, which is one of its modelling strength. Second, we're sampling from two distributions, which requires a fair amount of inference.

The second approach tackles both these issues. Using the ability to backward sample from $p_n()$, we sample $y' \sim p_n(f(\mathbf{x}) \mid y)$. We then make the GP observe y' and return as the particle's likelihood the likelihood returned by the GP. Once again, to allow for a small error from the sampling, we assume y' is normally distributed around $f(\mathbf{x})$ with variance σ_n^2. However, in this case, it can simply be integrated in the GP's covariance function as noise.

I find it is sometimes possible to empirically construct an approximation of this distribution $p_n(y'_t \mid y_t)$. The next subsection shows how I construct this distribution in the context of single threaded programs.

4.4.2 Empirically constructing the noise distribution

This subsection presents a method to build an empirical distribution of the noise in measured runtimes in the context of the execution of single threaded programs. I present a data structure called `OSTimeSampler` which lets us sample from the posterior of the noise distribution. Given a measured runtime y, `OSTimeSampler` can sample an average execution time which could have generated y.

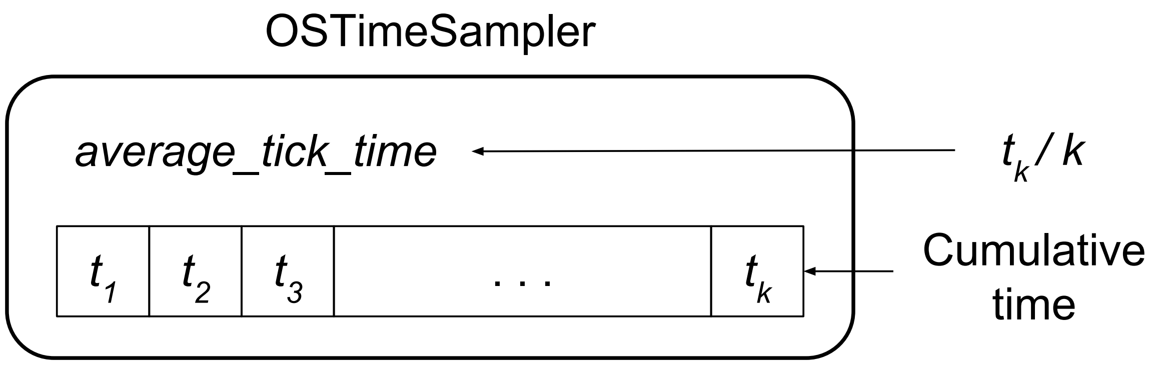


Figure 4.4: The `OSTimeSampler` architecture.

Figure 4.4 summarises the architecture of `OSTimeSampler`. There are two phases, a *load* phase in which `OSTimeSampler` gathers data by measuring the runtime of computations, and a *sample* phase in which we will be able to sample from the distribution. The load phase should be executed in the same context as the experiments which we would like to perform inference over.

In the load phase, `OSTimeSampler` iteratively performs a fixed short computation. In the actual implementation, the computations consists of a thousand floating point divisions. It performs this computation k times, where k is an input parameter. At the end of each iteration i, it stores t_i, the time that elapsed since the load phase started. I call this a "tick". By performing this procedure over a long enough time, we can get a good estimate of the mean duration of each tick as t_k/k.

In the sample phase, `OSTimeSampler` can be sampled from. This procedure is shown in Algorithm 4.3 and takes as input a measured runtime t. `OSTimeSampler` selects a random tick τ on the array with associated time t_τ. From there, the goal is to estimate, from when tick τ happened, how much work was performed in the following time t.

To do so, `OSTimeSampler` finds the last tick τ' with time smaller than $t_\tau + t$ using a binary search. It then linearly interpolates between ticks τ' and $\tau' + 1$ to account for the difference between t and $t_{\tau'} - t_\tau$. This yields an estimate for the quantity of computational work that occurred over the time t.

Finally, `OSTimeSampler` returns the average time needed to execute this computation by using its computed average tick time. From a Bayesian point of view, this procedure samples from the posterior distribution of average times given t, with a uniform prior over average times.

In the context of semi-parametric models implemented in Probabilistic-C++, `OSTimeSampler` can be used to pre-process a measured execution time by each model object independently. Then, as is done in semi-parametric models, we subtract the time predicted by the parametric model from this sampled average time. We feed this difference to the GP. Finally,

Algorithm 4.3 Sampling procedure of `OSTimeSampler`.

```
Input: Measured time $t$ of the computation
1: Sample $\tau \sim \mathcal{U}(1, k)$
2: Binary search greatest tick $\tau'$ such that $t_{\tau'} - t_\tau < t$
3: $work \leftarrow \tau' - \tau + \frac{t-(t_{\tau'}-t_\tau)}{t_{\tau'+1}-t_{\tau'}}$
4: Return $work \times average\_tick\_time$
```

we return the GP's likelihood for the measurement as the model object's likelihood of observing the data.

The SMC procedure will then assign low probabilities to particles which found this data unlikely, possibly due to a bad sample from the `OSTimeSampler`, and high probabilities to those which found the data likely. Over time, particles which selected bad `OSTimeSampler` samples will be deleted through resampling and others kept. This sort of inference over a single random draw per measurement is what SMC is most suited to. In the next subsection, I evaluate `OSTimeSampler` and show code examples of its use.

4.4.3 Evaluation

I evaluated the method proposed in this section by comparing the performance of a model which uses the sampler, against a model that does not. The models infer the average time needed to multiply two matrices A and B of size $l \times m$ and $m \times n$ respectively.

Data. I sampled 100 random pairs of matrices, with l, m and n each sampled at uniform random in the range [50, 500], and the content of the matrices being drawn at random between 0 and 1. All of my measurements were performed in the environment described at the beginning of this section: a four hyper-thread `m4.xlarge` EC2 instance with four other concurrent processes executing infinite bash loops. The distribution of Figure 4.3 is therefore representative of the computation noise.

After "warming up" the program for a while by executing some other random matrix multiplications, I measured the time to multiply each of the matrix pairs. Multiplications were performed using the Eigen library [GJ+10] in single threaded mode. I did some other random matrix multiplications between each pair to avoid the times being correlated. Most of the measured times lie between 0.1ms and 50ms.

The goal of inference is to deduce the average time to perform a matrix multiplication as a function of the matrices dimensions. Hence, it is useful to have an estimate of these average times in order to measure the error of the models. I therefore subsequently repeated the same experiment 100 times to have an accurate estimate of the average time needed to multiply each matrix pair. Finally, I executed the load phase of `OSTimeSampler` in this environment.

Model. I designed a semi-parametric model which attempts to predict the time to

```
double observe(int l, int m, int n, double time) {
  time = os_time.backwards_sample_milisecs(time);
  time -= parametric(l, m, n);
  return gp.observe({l, m, n}, time);
}
```

Listing 4.12: Using `OSTimeSampler` in a model class.

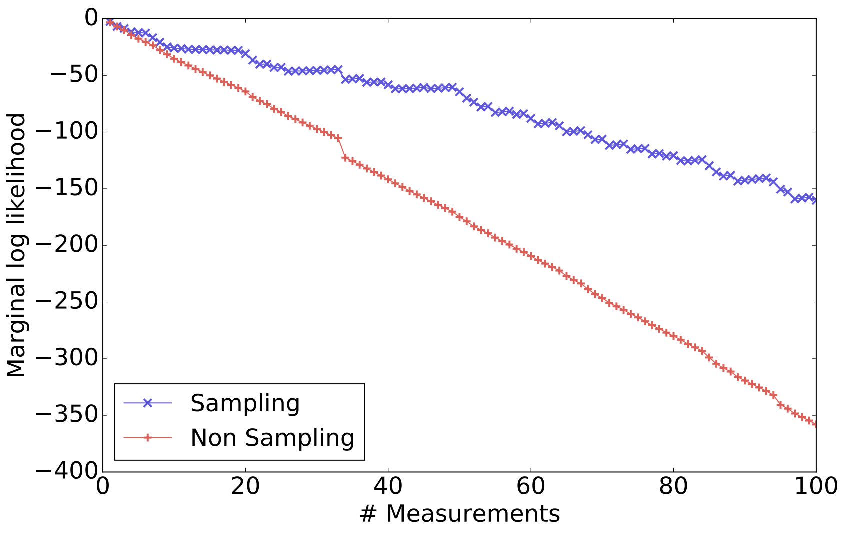


Figure 4.5: Evolution of the marginal log-likelihood as both models observe the data.

multiply the two matrices as a function of l, m and n. The parametric part of the model contains a single parameter α. It predicts the execution time as αlmn. The non-parametric part is a Gaussian process with 0 mean and standard deviation σ being inferred by the probabilistic program. Its length-scale ℓ is the same for all three dimensions and is inferred too. Finally, the noise per measurement σ_n is inferred as well.

I compared two versions of this model. First, the non-sampling version in which the runtime measurement is not preprocessed. This means the noise is modelled as Gaussian. Second, the sampling version in which I sampled an average execution time using `OSTimeSampler`. All of the parameters prior distributions are identical except for the noise of the Gaussian process σ_n. In the first version the Gaussian process is responsible for modelling all of the noise, and the posterior distribution of σ_n is centred around 8.0ms. In the second version, as `OSTimeSampler` models most of the noise, the values of the posterior of σ_n tend to lie near 0.15ms. I therefore adapted the prior distributions accordingly.

To show the simplicity of using `OSTimeSampler`, I show the implementation of the `observe` function of the second model in Listing 4.12. I use each model in a `ProbEngine` with 100,000 particles. In the non-sampling version, I take advantage of the deterministic nature of the observes by always using `observe_deterministic`, hence no resampling is performed.

Figure 4.5 shows the evolution of the marginal log-likelihood as the data is measured.

	Sampling	Non-Sampling
RMSE	0.438ms	4.58ms

Table 4.2: Root mean squared error of the sampling and non-sampling versions on the average execution times.

Initially, both models were similarly surprised by the data. But as more measurements were made available, the sampling version started understanding the distribution and hence explained the data better. After all measurements were processed, the sampling version had a much higher log-likelihood.

Once both models were been trained, I queried them on each item from the training set, and compare their inferred average time with the estimated average time. This is analogous to *smoothing* in particle filtering. Table 4.2 shows the root mean squared error of both trained models.

The sampling model is significantly more accurate than the non-sampling version. This is because, in order to explain the outliers, the non-sampling model is forced to use a very large noise, above 8.0ms, which is greater than most measurements. Hence, it ends up barely learning from the data. In comparison, the sampling version infers a very small noise, around 0.15ms, and explains the rest via `OSTimeSampler`.

The environment which I used to generate the data is extremely noisy and hence showcases `OSTimeSampler` well. In more stable cases, such as an otherwise free machine, I find `OSTimeSampler` still brings an improvement over the naive non-sampling version, albeit not as clearly. In more complex distributed environments, `OSTimeSampler` is not applicable and I resort to repeating runs.

`OSTimeSampler` was originally designed in the context of the sort case study. However it turned out that, due to the cost of the numerical optimisation stage of the Bayesian optimisation, it was more efficient to repeat brief measurements in order to get an accurate measurement. Hence, none of the case studies evaluated in Chapter 7 make use of it.

4.5 Fast non-parametric models

As mentioned in Section 4.3.1, non-parametric models are crucial to Bayesian optimisation. They let us accurately model the region around the optimum. Gaussian processes are the non-parametric tool of choice when performing Bayesian regression. However, one of the difficulties of Gaussian Processes is their computational complexity, as discussed in Section 2.1.2. After n observations, the cost of making a prediction is $\mathcal{O}(n^2)$. In traditional Bayesian optimisation, this is often acceptable as the optimisation performs one observation per iteration and runs for dozens of iterations. In our context, however, this cost can be prohibitive as each iteration may yield thousands of observations.

In this section, I present a novel treed Gaussian process model which is well suited to the Probabilistic-C++ framework. While not statistically rigorous, it performs well in practice.

Section 4.5.1 lists the desirable requirements from a non-parametric model and discusses why the existent extensions to Gaussian processes, highlighted in Section 2.1.4, were insufficient in this context. I then present my proposed approach (Section 4.5.2) and evaluate it (Section 4.5.3). Finally, I present two extensions which tend to make the model more robust (Section 4.5.4).

4.5.1 Requirements

Ideally, a non-parametric model would have the following properties:

- **Ability to compute the log-likelihood of observations.** When an observation is performed in Probabilistic-C++, each model object needs to return a log-likelihood quantifying how "surprised" it is to see that observation. This log-likelihood is what lets us establish which of the model objects best fit the observations. Non-parametric probabilistic models which are not Bayesian, such as random forests [Bre01], tend to not have a direct way of calculating this likelihood. This makes their application to Probabilistic-C++ difficult.
- **Have the memoisation property.** If we query a model on or near a point for which we have an observation, we want to guarantee that the corresponding prediction will match our observation. This requirement discards sparse Gaussian process approaches [SWL03, SG05, Tit09] which approximate a Gaussian process by fitting it on a restricted number of synthetic data points.
- **Sub-linear prediction cost.** In order to be useful in a general context, users should not have to reduce the amount of data they feed the model in order to reduce its computational complexity. The model should have a sub-linear prediction cost with respect to the number of measurements. This discards expert approaches [RG01, DN15]. Expert methods train multiple Gaussian processes on different subsets of the data. At prediction time, each of the experts is queried and their predictions are then weighted based on their confidence on the specific input. While this is under $\mathcal{O}(n^2)$, it is still at least $\mathcal{O}(n)$ as the input will be compared to each training data point.
- **Sub-linear observation cost.** In Bayesian optimisation, each iteration leads to new measurements being added into the model. In order to be practical, the time spent per observation by our model should not grow with the number observations performed so far. This discards the treed Gaussian process models proposed by

Gramacy et al [GL07]. These models use a decision tree data structure with Gaussian processes at the leaves. In order to integrate over the many possible trees, they use an Markov Chain Monte Carlo (MCMC) step which iteratively mutates the tree. This produces many decision trees, from which predictions are averaged out. Because this procedure is data-dependent, it must be executed before making predictions each time new observations are added. Performing MCMC on the entire decision tree is at least $\mathcal{O}(n)$ and hence not practical in our setting.

4.5.2 Proposed approach

My approach is inspired by the treed-GP of Gramacy et al. In our context, the non parametric model is used as part of a probabilistic programming library. This means we could achieve the same property of integrating among many possible trees by allowing a distribution over trees as part of the probabilistic framework. The model works as follows.

I iteratively build a decision tree with GPs at the leaves. When a new input is observed, it is propagated down the decision tree and added to the GP at its leaf. A *threshold* parameter specifies the maximum number of data points that can be stored in a leaf. When a leaf grows beyond this threshold, it is recursively split it across one of the input dimensions. I pick the dimension which sees a maximum spread of the data points once normalised with the Gaussian process' linear scales. Since Probabilistic-C++ allows users to provide a prior over those linear scales, different particles will split the data across different dimensions.

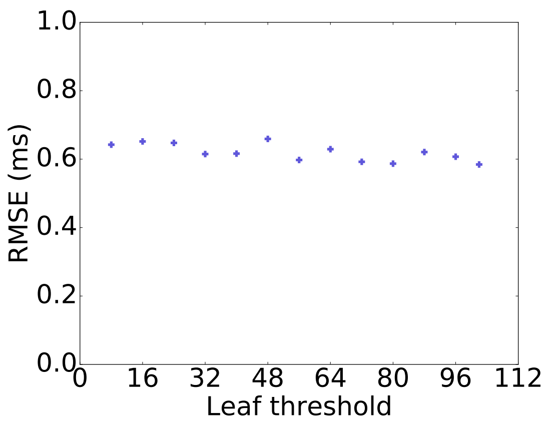
A few notable aspects and limitations of this procedure are:

- The data is no longer exchangeable with respect to the model. The order in which the data is observed by the model will have an influence on the dimensions used for splitting.

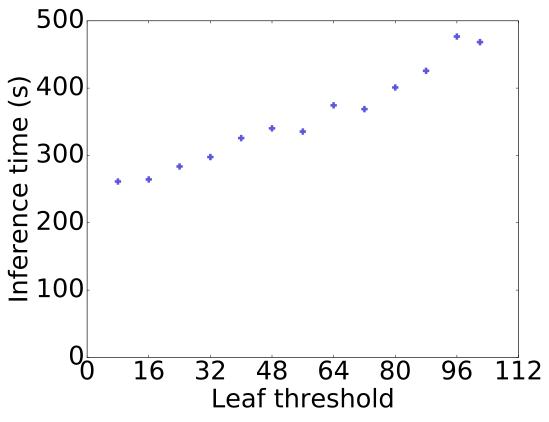
- The resulting model resembles a random forest. Each particle uses a decision tree to make predictions. An overall prediction is produced by averaging over all particles. However, the splits are deterministic given the GP's linear scales. Hence, if the model has degeneracy, and only one of the original particles has a non-zero probability, all trees will have the same structure.

- If the threshold is larger than the number of measurements, the single leaf never gets split. This leads to traditional Gaussian process regression.

The next subsection evaluates this approach.



(a) Error on the test data.



(b) Computational performance.

Figure 4.6: Evaluation of the treed GP on the matrix multiplication data. The rightmost point fits a single GP.

4.5.3 Evaluation

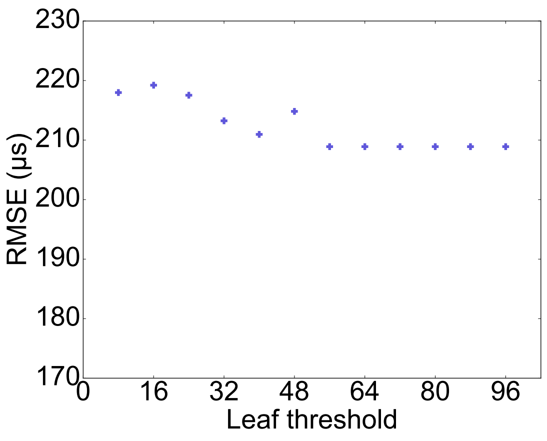
I evaluated this treed-GP implementation in two contexts:

1. The matrix multiplication data presented in the previous section. There were originally 100 measurements, which I used as training data. I generated another 100 average matrix multiplication measurements which I use as a test dataset. I used the same model as the one presented in the previous section apart from a single difference: I draw the three length scales hyper-parameters of the GP individually so that different particles split the data across different dimensions.

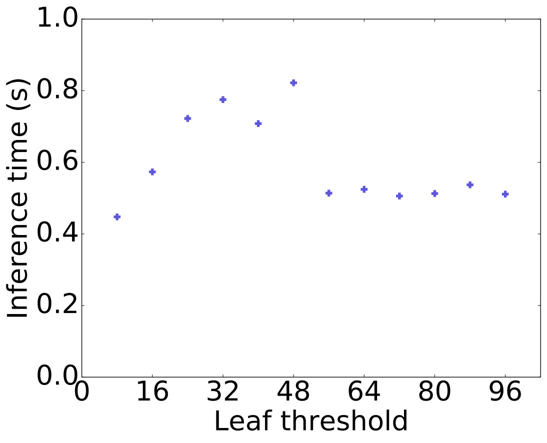
2. A dataset from the sort case study. Each iteration of the Bayesian optimisation, I measured the individual times to sort 1000 arrays. Here I selected the data generated from a single iteration. I set the last 500 measurements to be the test set. I vary the size of the training set to evaluate its impact on performance. The model is semi-parametric and predicts each array's sort time. It does so using three inputs: *(i)* the array's length, *(ii)* the number of adjacent items in the array that are unordered and *(iii)* the `block_size` parameter of the sort procedure, part of the configuration space in the case study.

In both contexts I evaluated the impact of the threshold parameter on the time to perform inference, and on the root mean squared error on the test dataset. All runtimes were evaluated on an `m4.xlarge` EC2 instance.

Matrix multiplication data. Figure 4.6 shows root mean squared error of the model and the inference time. Interestingly, the threshold value seems to have had little importance on the predictive performance of the model. In particular, the highest threshold is greater than the size of the data and hence we are simply fitting a single Gaussian process. The resulting error is only slightly lower than the other models.

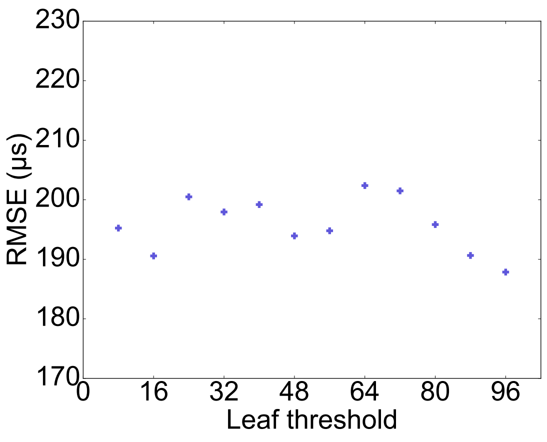


(a) Error on the test data.

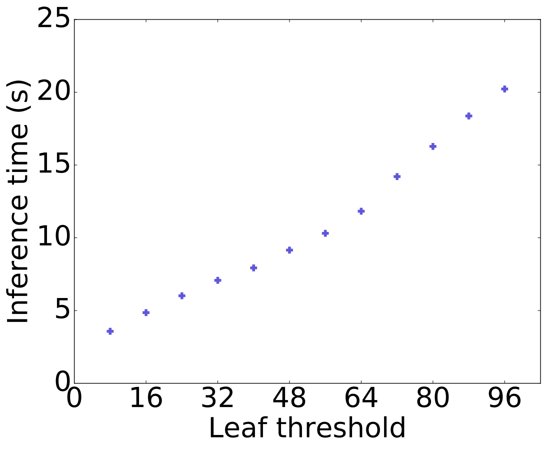


(b) Computational performance.

Figure 4.7: Evaluation of the treed GP on the sort case study data, trained with 50 samples. Note that all thresholds above 50 simply use a single Gaussian process.



(a) Error on the test data.



(b) Computational performance.

Figure 4.8: Evaluation of the treed GP on the sort case study data, trained with 400 samples.

Sort case study data. Figure 4.7 and 4.8 show the results on the sort data for a training set of size 50 and 400 respectively. I only used 1000 particles in the probabilistic engine, which explains why the inference times are shorter than for the matrix multiplication case. Interestingly, the threshold parameter had more of an impact on inference time than it did for matrix multiplication. I suspect this is because the smaller number of particles lead to more memory locality, making computational performance more important.

Once again, the deviation in resulting error are small. We would expect that with fewer measurements, the better generalisation properties of the GP would show. However, the treed GPs seem to perform reasonably well.

4.5.4 Extensions

I found the following further techniques led to more robust models:

- **Overlapped split.** When splitting a leaf into two after it has reached the threshold number of observations, I duplicate the k data points closest to the border on either side of the branch, where k is a parameter. I find this reduces the discontinuity that occurs across branches.

- **Not splitting at the median.** Often when performing optimisations, observations have a distribution that is centred near the optimum. When splitting a leaf into two, the region that loses the most precision is around the border. Therefore, if we were to set the border at the median, we would lose prediction power where it is most important. Instead, I set the border to be on the data point at index:

  $$\frac{3-\sqrt{5}}{2} \times \mathit{threshold} \approx 0.382 \times \mathit{threshold}.$$

  This has the property that after a split followed by another split on the same dimension of the initially larger right leaf, the middle partition will be centred on the median.

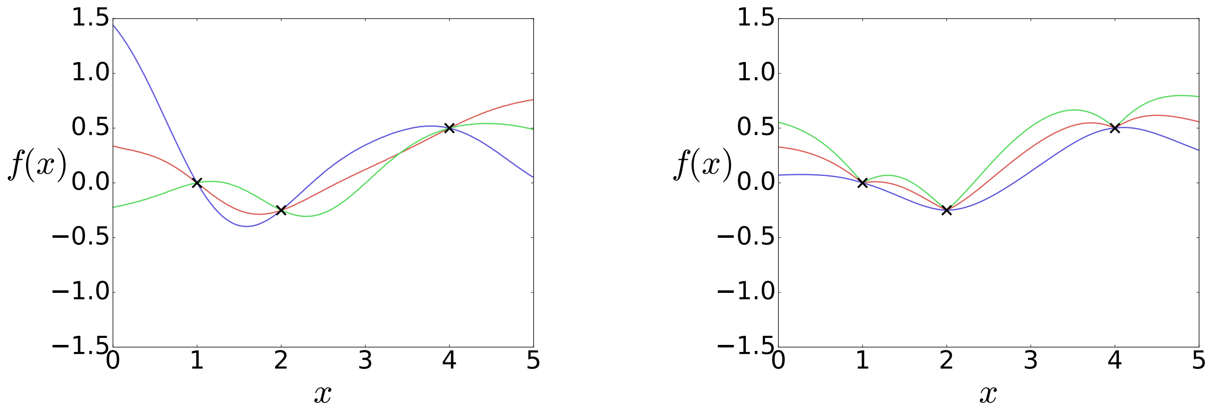
In practice, I use this treed Gaussian process model by default when using a non-parametric models, with a threshold of 64 and an overlap of $k = 3$.

4.6 Sampling from a model

In the next two chapters, I will present techniques that rely on our ability to sample a single instance of the model from a `ProbEngine`. This will allow techniques similar to Thomson sampling. In this section, I briefly explain how I achieve this in the context of Probabilistic-C++.

Sampling from a parametric model is simple. We select one of the particles with probability weighted by their likelihood. In Probabilistic-C++ this can be achieved by calling `ProbEngine::single_particle_engine()` which will select one of the particles at random.

If the model class of the probabilistic model also contains a non-parametric model, then we must also sample a single response surface from that model. Sampling from the posterior of a Gaussian process is too computationally expensive to be practical in multi-dimensional domains. One way to approximate a sample would be to use the approximate spectral sampling techniques introduced by Hernández-Lobato et al. [HLHG14]. I use a simpler approach which works well in practice inspired by the upper confidence bound



(a) True samples. **(b)** Approximated samples.

Figure 4.9: True samples and approximated samples from the posterior of a Gaussian process after three measurements.

acquisition function [SKSK10]. Recall from Section 2.2.2 that the upper confidence bound of a Gaussian process is written as:

$$\alpha_{\text{UCB}}(\mathbf{x}) = \mu(\mathbf{x}) + \kappa\sigma(\mathbf{x}) \tag{4.2}$$

where κ is a positive parameter. This gives a response surface that is higher than the mean. When sampling, I will want the surface to be sometimes higher and sometimes lower than the mean. I achieve this by simply picking κ from a standard normal distribution $\mathcal{N}(0, 1)$. Figure 4.9 compares samples drawn from the true posterior to approximated samples.

A desirable property of this approach is that the resulting distribution of an input $\mathbf{x}$ is identical to its true distribution. However, unlike true samples from the Gaussian process, this response surface is not smooth at observed measurements, and tends to be flatter than true samples from the Gaussian process. When using a treed Gaussian process, as presented in the previous section, I sample a single value for κ for each of the Gaussian processes at the leaves. I did experiment with using a different sample of κ for each leaf, but the resulting sampled distributions were too discontinuous to be practical.

Hence, when sampling from a Probabilistic-C++ semi-parametric model, I sample a single particle from the `ProbEngine`. Then, for each non-parametric model in the particle (usually one), I sample a single value κ and subsequently use Equation 4.2 to make predictions. With my Gaussian process implementation, this is automated by using the method `GaussianProcess::set_sampling()` or `TreedGP::set_sampling()`.

4.7 Summary

This chapter presented a set of techniques and abstractions designed for a developer to build a probabilistic model of their system's behaviour. The core abstraction is the one offered by the Probabilistic-C++ programming library. It allows a developer to specify a prior distribution of their model and perform inference on it via observations.

In the context of a structured Bayesian optimisation, I argued that the best way to develop a model was to construct a set of independent semi-parametric models. They combine a parametric model, written by the developer, which can capture the overall trend of the data, with a non-parametric model, such as a Gaussian process, which will interpolate between measurements. One can design multiple semi-parametric models, representing different components of the system being modelled, and expose their independence to Probabilistic-C++ by performing inference on them independently.

Finally, I presented three techniques tackling different problems that may occur when performing structured Bayesian optimisation on a program's performance. First, I showed how to tackle the long tail in the noise distribution of brief runtime measurements. Then, I presented a treed Gaussian process model which, unlike simple Gaussian processes, can handle very large numbers of measurements. Finally, I discussed how to sample from a probabilistic model, which will be useful for the numerical optimisation stage of a structured Bayesian optimisation.

CHAPTER 5

BOAT's optimisation scheduling abstraction and its use for decompositions

With each iteration of a Bayesian optimisation, a *numerical optimisation* is performed. The goal is to find a configuration deemed promising by the model. In traditional Bayesian optimisation, an off-the-shelf optimiser will typically be used. Unfortunately, for complex optimisation problems, such off-the-shelf optimisers will often fail to converge. This is a significant issue, if the numerical optimisation fails to find good configurations, the Bayesian optimisation itself will not converge. This chapter is jointly concerned with two related topics. First, techniques which exploit the domain specific structure of a numerical optimisation problem to improve convergence. Second, the implementation of these techniques in the BOAT framework. It makes the following contributions:

- I present BOAT's optimisation scheduling abstraction. Its key feature is the ability to nest optimisations within one another. This formalises an often used pattern of executing optimisations as part of another optimisation (Section 5.1).
- I show BOAT's interface to off-the-shelf numerical optimisation methods, which can be used by default in a numerical optimisation. I quantify the limits of these methods by using two synthetic benchmarks inspired by the case studies presented in this dissertation (Section 5.2).
- I discuss how *decompositions methods*, which are often used in numerical optimisation problems, can be exploited to tackle the limits of off-the-shelf methods in the context of Bayesian optimisation. I show how decompositions are also easy to express using BOAT's optimisation abstraction (Section 5.3).

- Even when using decompositions, the convergence of the numerical optimisation may still be slow, especially for large configuration spaces. I present two new techniques which can be used to improve this convergence. The first one is based on structured Bayesian optimisation. Using decompositions can lead to performing optimisations with expensive objective functions – the objective function will itself consist of other optimisations. Hence, it can be useful to reduce the number of iterations at the cost of a higher overhead per iteration. This is exactly the trade-off offered by Bayesian optimisation and I show how it can be exploited in this context (Section 5.4). The second technique builds on top of the first one and uses an approach inspired by reinforcement learning. This allows the models used in Bayesian optimisation to "remember" the shape of the objective function between subsequent optimisations (Section 5.5).

- I empirically compare the performance of these new techniques against traditional decompositions in the context of the sort case study (Section 5.6).

Note that within this chapter, I always assume the role of the numerical optimisation is to find the configuration predicted best by the model. This is different from optimising an acquisition function, as presented in Section 2.2, which optimises a combination of exploration and exploitation. I will discuss in Chapter 6 how the methods presented in this chapter can still be used for exploration. In essence, the methods discussed here will be used to generate Thompson samples.

5.1 Optimisation scheduling abstraction

This section introduces the optimisation scheduling abstraction offered by BOAT which can be used to optimise BOAT configuration spaces, as introduced in Section 3.3.2. This abstraction will prove useful in a number of contexts.

I first discuss how to assign values to BOAT parameters (Section 5.1.1) and show how this can be used within a simple optimisation (Section 5.1.2). I then present the key feature of BOAT's optimisation abstraction: the ability to nest optimisations within one another (Section 5.1.3). Finally, I discuss the optimisation of a special type of parameters, `ParameterPtr`s which are BOAT's way of handling dependencies in the configuration space (Section 5.1.4).

5.1.1 Assigning values to parameters

A parameter can be assigned to and read from by using its `assign` and `value` member functions respectively. The following code exemplifies these two functions:

```
RangeParameter<double> a(0.0, 1.0);
SimpleOpt<double> opt;
opt.set_iteration_function(
    [&](){
      // a has no assigned value yet
      assert(!a.has_assignment());
      a.assign(uniform_real_distribution<>(0.0, 1.0)(generator));
      assert(a.has_assignment());
      return a.value() * (1.0 - a.value());
     });
opt.run_optimisation();

// Print the best value found
std::cout << a.value() << std::endl;
```

Listing 5.1: Simple optimisation example.

```
RangeParameter<double> param(0.0, 1.0);
param.assign(0.4);
std::cout << param.value() << std::endl; //Prints 0.4
```

If a parameter already has an assigned value, attempting to assign another value will result in an error. For example:

```
RangeParameter<double> param(0.0, 1.0);
param.assign(0.4);
param.assign(0.6); //Yields an error
```

Similarly, querying an unassigned parameter also yields an error:

```
RangeParameter<double> param(0.0, 1.0);
std::cout << param.value() << std::endl; //Yields an error
```

5.1.2 Simple optimisations

In practice, we want to use parameters for optimisations, and doing so involves iteratively assigning different values to them to find out which ones yield better outputs from the objective function. Listing 5.1 shows an optimisation which randomly samples values for a parameter a in the range $[0, 1]$ to minimise $a(1 - a)$.

At the beginning of each iteration, a appears to be unassigned. Hence, we can assign it a new value each iteration. Throughout iterations, the optimisation keeps track of the different values that were assigned to a and the corresponding result from the objective function. Once the optimisation terminates – by default after 100 iterations – a remains assigned with the value that generated the highest objective value. I now explain how this is implemented internally.

Each iteration of an optimisation, a *parameter instance* is implicitly created. Whenever a value is assigned to a parameter, the parameter instance of the current iteration is associated with it. The optimisation is responsible for managing the set of instances it creates. Once it returns, the instance that has yielded the best result of the objective function is *promoted*, and the other instances discarded. As a result, values assigned during this best iteration remain visible afterwards.

Reading values from previous iterations. Throughout an optimisation, it may be useful to be able to query previous parameter values. For example, simulated annealing, which I present in the next section, works by mutating one of the previously assigned configurations. The `ParameterInstanceID` of a previous iteration of an optimisation can be queried as

```
ParameterInstanceID id = opt.get_parameter_instance_id(iter_number);
```

The values that were assigned to parameters on those iterations are then accessible by calling `param.value(id)`. When calling `value()` without an argument, as presented before, we are implicitly passing down the current iteration's parameter instance.

Abstraction offered by an optimisation in BOAT. To summarise, an optimisation in BOAT offers the following abstraction. It implicitly takes as input a set of unassigned parameters. After the execution, the parameters are assigned with values which maximise the optimisation's objective function.

5.1.3 Nested optimisations

The key aspect of BOAT's optimisation abstraction is the ability to nest optimisations within one another. Say an optimisation was given the task of optimising a parameter a. This means it must assign a value to a each iteration. The optimisation may delegate this task to a *sub-optimisation*. The sub-optimisation will be executed entirely each iteration of the higher level optimisation.

Although simple, this pattern of sub-optimisations is frequent and used by a range of optimisation techniques. Formalising it therefore simplifies the implementation of a number of optimisation methods. Two examples of nested optimisation already discussed in this dissertation are:

- **Bayesian optimisations** which performs a numerical optimisation each iteration to find promising values.
- **Portfolio acquisition functions**, described in Section 2.2.2, which make the numerical optimisation stage of the Bayesian optimisation perform multiple independent optimisations using different acquisition functions. At the end of the numerical optimisation, the parameters are left assigned with the values that maximise a meta-criterion.

```
void optim_single_param(RangeParameter<double>& param){
  SimpleOpt<double> opt;
  opt.set_iteration_function(
      [&](){
        param.assign(uniform_real_distribution<>(0.0, 1.0)(generator));
        return param.value() * (1.0 - param.value());
      });
  opt.run_optimisation();
}
int main(){
  RangeParameter<double> a(0.0, 1.0), b(0.0, 1.0);
  SimpleOpt<double> opt;
  opt.set_iteration_function(
    [&](){
      optim_single_param(a);
      optim_single_param(b);
      return a.value() * (1.0 - a.value()) * b.value() * (1.0 - b.value());
    });
  opt.run_optimisation();
}
```

Listing 5.2: Nested optimisations example.

Furthermore, I will use this abstraction extensively later in this chapter to implement decompositions.

I here present an example of this behaviour, although it is too simplistic to be realistic. In Listing 5.2, I optimise two values a and b, each in the range $[0, 1]$, to minimise the objective function $a(1.0 - a)b(1.0 - b)$. The task of the high level optimisation is to assign good values to a and b. To do so, each iteration, it delegates this task to two sub-optimisations which optimise a and b individually.

Figure 5.1 shows how BOAT handles this internally. For each iteration, the corresponding optimisation keeps track of the parameter values that were assigned to the parameters, and the utility that was returned. When a sub-optimisation is executed, the task of tracking parameter values and their associated utilities is delegated to that sub-optimisation. This is the case in the upper left diagram of Figure 5.1, in which sub-optimisation 1 keeps track of the different parameter values that have been assigned to a.

Once the sub-optimisation returns, it selects the iteration that yielded the best utility. The associated parameter values are then promoted and considered part of the higher-level optimisation. This is shown in the lower left diagram of Figure 5.1 in which the best value found for a within sub-optimisation 1 got promoted.

Finally, the lower right diagram shows the second sub-optimisation being executed. Because a has been assigned a value through sub-optimisation 1, it appears as assigned. This means that calling `a.value()` within sub-optimisation 2 would return a_3. If instead, `a.value()` was called before sub-optimisation 1, before the upper left diagram, it would

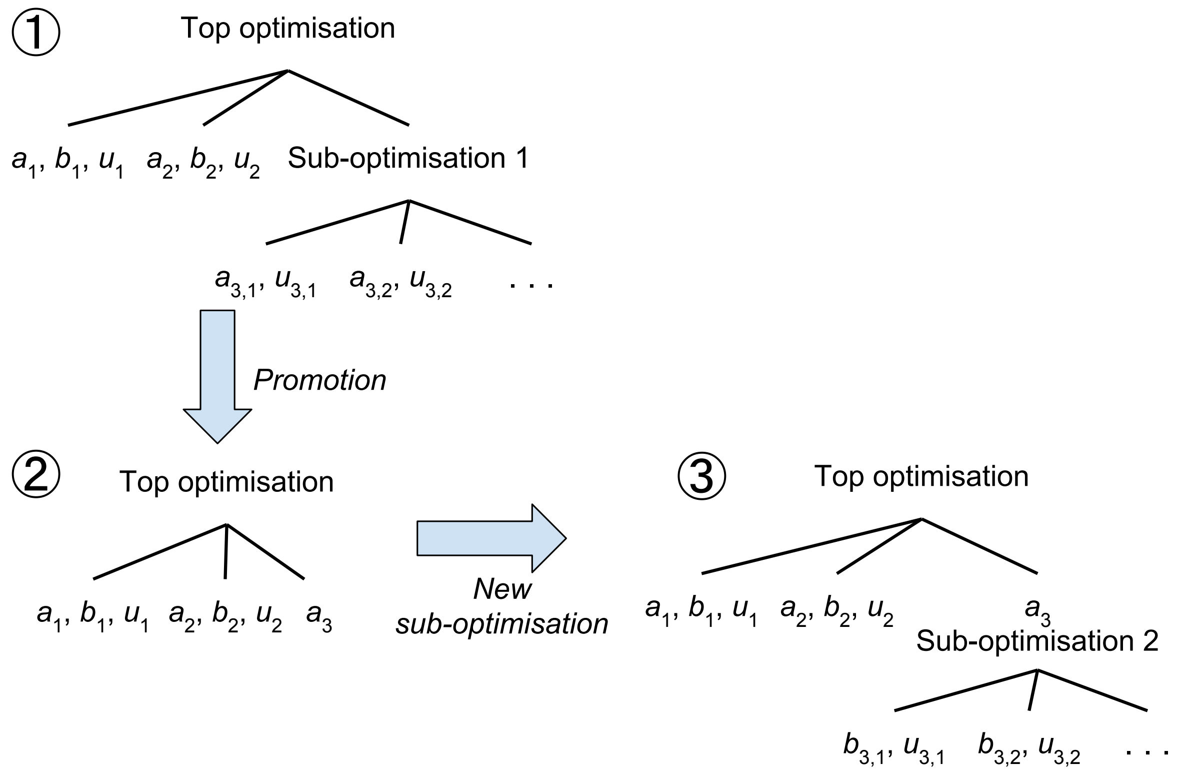


Figure 5.1: Nested optimisation procedure.

result in an error as a was not assigned to yet.

Throughout this dissertation, I often use a diagrammatic representation of sub-optimisations. Figure 5.2 shows the representation of the optimisation of a and b, with both sub-optimisations.

5.1.4 Parameter pointers

Recall from Section 3.3.2 that `ParameterPtr` objects are designed to hold pointer to parameters or sets of parameters whose existence or domain is conditional. From the point of view of BOAT, `ParameterPtr`s are another special type of parameters and go through the same process of assignment and promotion. Unlike regular parameters, `ParameterPtr`s can only be assigned to via the `parameter_function()` of a `ParameterSpace` object. This is so that the optimisation procedure, which assigns values to parameters, can be decoupled from the domain properties.

The `parameter_function()` of `ParameterSpace` objects is executed lazily. When an optimisation calls `ParameterPtr::deref()` on a `ParameterPtr`, either it already has an assignment or it does not. If it does, `deref()` simply returns a reference to the corresponding content. If it does not, the `parameter_function()` of the enclosing `ParameterSpace` object is executed up to the point where the content of this `ParameterPtr` is created. The

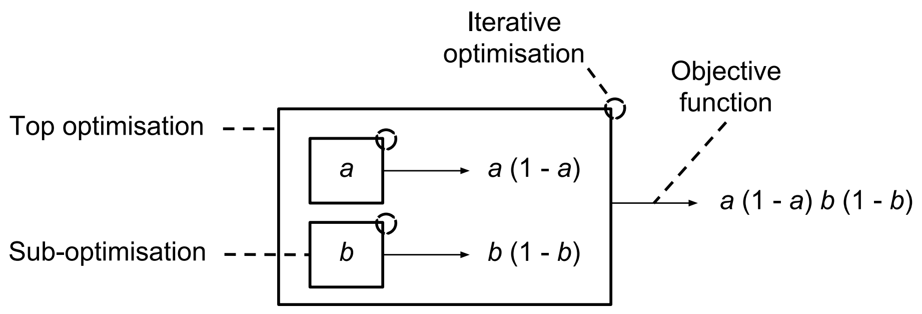


Figure 5.2: Diagram of an optimisation with two sub-optimisations.

`ParameterPtr` is then assigned with a pointer to this content. Finally, the content itself is returned.

A notable implementation detail is that parameter functions are executed via *coroutines* [Con63], which are program executions which can be interrupted and resumed, using Boost coroutine's library [Boo16]. When a previously called `parameter_function()` is called again to create the content of another `ParameterPtr`, BOAT performs the minimum amount of work necessary. The previous execution of the `parameter_function()` is simply resumed up to the point where it assigns content to the `ParameterPtr` being dereferenced.

In conclusion, using BOAT's optimisation abstraction, a developer can implement nested optimisations in a structured way. BOAT hides the implementation details of managing the values assigned to the parameters at each iteration. This allows the developer to focus on the core properties of the optimisation, such as its objective function, or the sub-optimisations it uses. The next section shows BOAT's interface to off-the-shelf optimisation routines.

5.2 Interface to numerical optimisation routines

The previous section showed optimisations which randomly sampled values each iteration. This is extremely inefficient. This section presents BOAT's interface to numerical optimisation routines. I show how to use off-the-shelf optimisation algorithms in Section 5.2.1. In parameter spaces with dependencies, these optimisers may not be applicable. I therefore present how to implement simulated annealing optimisations which are extremely general (Section 5.2.2). Finally, I quantify the capabilities of these generic optimisation methods (Section 5.2.3).

```
RangeParameter<double> a(0.0, 1.0), b(0.0, 1.0);
NLOpt<double> opt(a,b);
opt.set_utility_function(
    [&](){
      assert(a.has_assignment() && b.has_assignement());
      return a.value() * (1.0 - a.value() - b.value())
     });
opt.set_max_num_iterations(10000);
opt.run_optimisation();
```

Listing 5.3: Using the DIRECT optimisation algorithm in BOAT.

5.2.1 Off-the-shelf optimisers

BOAT offers an interface to the NLOpt toolbox [Joh14], as well as an implementation of the CMA-ES algorithm [HO01]. These optimisations take as part of their constructor the list of parameters they will be optimising. The code in Listing 5.3 optimises two parameters using the DIRECT algorithm [JPS93], which is used by default when using the NLOpt package. Using the CMA-ES algorithm can be done using the similar `CMAESOpt` class. In practice, I find that the DIRECT algorithm yields better performance than other off-the-shelf optimisation algorithms I have tried, and I therefore use it by default.

Limitations DIRECT and CMA-ES are well engineered algorithms and should be used whenever possible. However, they can only be applied in the domain $\mathbf{x} \in \mathbb{R}^d$. In more complex settings, such as ones with categorical parameters, or parameters dependencies, they cannot be applied. BOAT offers no off-the-shelf method to optimise these complex configuration spaces. Instead, it offers the skeleton of a simulated annealing optimisation which I describe in the next subsection.

5.2.2 Simulated annealing

This subsection presents the implementation and use of simulated annealing in BOAT. Simulated annealing [KGV83] is a stochastic algorithm for black-box optimisation inspired by statistical physics. It models a particle moving according to the *Boltzmann distribution*, which specifies the probability of being in state $\mathbf{x}$ as:

$$p(\mathbf{x}) \propto \exp(-f(\mathbf{x})/T)$$

where $f(\mathbf{x})$ is the energy of $\mathbf{x}$, which we try to minimise, and T is the temperature. In simulated annealing, we perform a random walk on that distribution and progressively lower the temperature. As T approaches 0, the particle spends more time in states of low energy.

To do a random walk, after k iterations, we propose a new state near $\mathbf{x}_k$. For example, if

```cpp
void proposal(const SortNode& from, SortNode& to, ParameterInstanceID id){
  // Mutate the is_leaf parameter
  bool from_is_leaf = from.is_leaf.value(id);
  bool to_is_leaf = from_is_leaf ? bernoulli_draw(0.9) :
                                   bernoulli_draw(0.1);
  to.is_leaf.assign(to_is_leaf);

  // If from_is_leaf == to_is_leaf mutate the rest of the tree recursively
  // Otherwise, generate it randomly
  ...
}
/*----------------------------------------------------------------------*/
// Body of a function executing a simulated annealing optimisation

SortNode sort_param;
SimulatedAnnealingOpt<double> opt;
opt.set_proposal_function([&](ParameterInstanceID id){
  proposal(sort_param, sort_param, id)
});
```

Listing 5.4: Using simulated annealing in BOAT.

$\mathbf{x}_k \in \mathbb{R}^d$, a typical proposal is $\mathbf{x}' = \mathbf{x}_k + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{\Sigma})$. Sometimes but not always, the step size decreases with the temperature. In our example, we could use $\boldsymbol{\Sigma} = T\mathbf{I}$. With the proposed state we compute

$$\alpha = \exp((f(\mathbf{x}_k) - f(\mathbf{x}'))/T).$$

We then accept the proposal with probability $\min(1, \alpha)$, in which case $\mathbf{x}_{k+1} \leftarrow \mathbf{x}'$, or we reject it and remain in the same state $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k$.

The rate at which the temperature is lowered is the *cooling schedule*. For example, an exponential cooling schedule is often used: $T_{k+1} \leftarrow CT_k$ where $C < 1.0$. Typically, $C = 0.95$. It has been shown that if the cooling schedule is sufficiently slow, the optimisation is guaranteed to find the global optimum [KGV83].

The reason I use simulated annealing as the generic optimisation method for complex domains is its simplicity. In order to use it on their problem, developers have to specify two functions:

- The *initial value* function which assigns parameters their initial value, and
- The *proposal* function which assigns parameters value from a proposal distribution, given the value of parameters from a previous iteration.

For example, Listing 5.4 shows part of the implementation of simulated annealing in BOAT to optimise a sort decision tree, as presented in Section 3.3.2. The top of the listing shows the proposal function, and the bottom shows its exposition to the `SimulatedAnnealingOpt`. As input, the proposal function takes a `ParameterInstanceID`, which is used to retrieve the values assigned to the parameters at the previous state. It then assigns new values to the parameters drawn from the proposal distribution.

There are other more elaborate methods to optimise complex configuration spaces, such as evolutionary algorithms. The reason BOAT includes simulated annealing is its simplicity for users. Most of the work consists of implementing the proposal distribution. In contrast, evolutionary algorithms require the implementation of more cumbersome crossover functions. They may perform better, but will still be inherently limited by their black box nature. Hence, domain specific structure is favoured as a way to help the optimisation converge, as I will show in the next section. The next subsection quantifies the capabilities and the limits of the optimisation algorithms I have presented so far.

5.2.3 Quantifying the limits of off-the-shelf numerical optimisers

For problems with a large number of dimensions, off-the-shelf optimisers may fail to find high quality values in a reasonable number of iterations. This is famously due to the *curse of dimensionality* – as the number of dimensions grows, the problem becomes exponentially hard. This subsection attempts to quantify the scale at which we can expect these optimisers to work. I benchmarked the performance of three algorithms: the DIRECT algorithm [JPS93], the CMA-ES algorithm [HO01] and simulated annealing [KGV83]. I did so using two synthetic benchmarks inspired from the case studies tackled in Chapter 7.

In both benchmarks, I computed the optimal configuration $\mathbf{x}_*$. I evaluated the performance of an optimisation by comparing its best found configuration with the optimal one. An optimisation was considered successful if its best configuration $\mathbf{x}$ is within 1% of the best configuration:

$$f(\mathbf{x}) \leq 1.01 \times f(\mathbf{x}_*)$$

In practice, I found optimisation algorithms tend to either converge after a low number of iterations (<10,000) or not converge at all. I evaluated the performance of the different algorithms after 100,000 iterations.

In the first benchmark, the optimisation balanced a computational load onto different simulated workers, each of which had different computational performance. The overall performance was the one of the slowest worker. The second benchmark optimised the parameters of a decision tree to make it reproduce its input.

Load Balance benchmark: For a given dimension d, I drew a uniform random vector **speeds** in the continuous space $[0.1, 1]^d$:

$$\mathbf{speeds} = speeds_1, \ldots, speeds_d \sim U(0.1, 1)^d$$

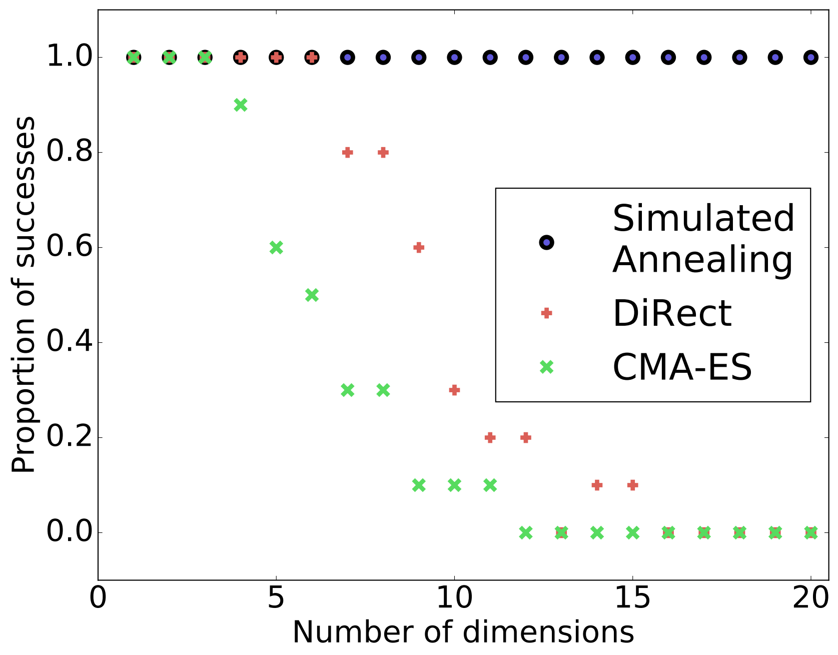


Figure 5.3: Evaluation of the performance the optimisation algorithms on the load balance benchmark. Each dimension was evaluated over 10 run.

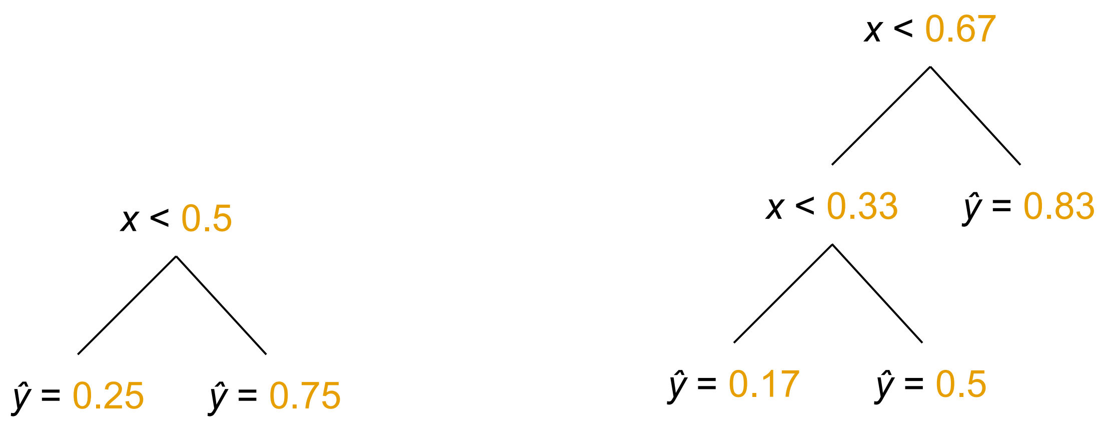
This conceptually represents the individual processing speed of the d workers. Consider the task of balancing work to minimise the maximum individual processing time. That is, given a vector $\mathbf{x} \in [0,1]^d$, we minimise:

$$f(\mathbf{x}) = \frac{1}{\sum_i x_i} \max_i (x_i / \mathit{speeds}_i).$$

The $\frac{1}{\sum_i x_i}$ term normalises the total amount of work. The value of x_i therefore represents the relative load assigned to worker i. For a given vector **speeds**, the optimal value of $\mathbf{x}$ is any vector along the line $\mathbf{x} = \alpha \times \mathbf{speeds}$ that lies in the domain $[0,1]^d$. Hence, I set $\mathbf{x}_* \leftarrow \mathbf{speeds}$.

Evaluation procedure: I evaluated all three algorithms via the BOAT framework. To evaluate an algorithm, I performed 100,000 iterations and observed whether the success criterion had been reached. Each optimisation took roughly a minute to return. Figure 5.3 shows the proportion of time the success criterion was reached as a function of the dimension for all three algorithms.

We see that while simulated annealing always converged, DiRect and CMA-ES often failed beyond 10 dimensions. Although both are designed to be global optimisation algorithms, it appears they somehow got stuck in regions of low performance and failed to continue exploring. The optimisations which succeeded typically converged within 1500 iterations, a point after which others ceased to improve. Simulate annealing is well suited to this benchmark as it performs a local search. Reducing a single dimension, the one of the



(a) Three Dimensions **(b)** Five Dimensions

Figure 5.4: Structure of decision trees and optimal parameter values.

slowest worker, will improve performance.

Decision tree benchmark In this benchmark, I optimised the parameters of a binary decision tree to make it reproduce its input as closely as possible. The tree takes inputs drawn from a uniform distribution $x \sim U(0, 1)$. Each branch of the tree compares the value of x to a fixed parameter value and propagates x to one of its children accordingly. When x reaches a leaf l, the value $\hat{y}_l$ of the leaf is compared to x. I minimised the *squared residual* $(x - \hat{y}_l)^2$. The structure of the tree was fixed in advance. I always used *complete* trees, every level of the tree was filled except possibly for the last level, in which all nodes were as far left as possible. Figure 5.4 shows optimal parameter values for trees of dimension 3 and 5.

To evaluate the performance of a tree, I computed for each leaf l the range of inputs it received $[l_{min}, l_{max}]$. I then computed the integral of the squared residual of the leaf for that range:

$$l_{perf} = \int_{l_{min}}^{l_{max}} (x - \hat{y}_l)^2 dx.$$

The total performance of a tree was the sum of the leaves' performance: $\sum_l l_{perf}$.

Figure 5.5 shows the result. This benchmark is more difficult than the first one. Although both are convex, the "valley" of efficient implementations is much shallower in this one. This is because, in order to move from a good configuration to a better one, all parameters should be updated in synchronicity. In contrast, in the previous benchmark, one can reach the optimal value by mutating one parameter at a time.

The take away message from these two benchmarks is: one should not expect to achieve good results with off-the-shelf numerical optimisation software beyond five dimensions. The two benchmarks were simpler than most functions we would like to optimise. They were both continuous. Numerical optimisation methods typically perform much worse in discontinuous spaces. The benchmarks were also convex.

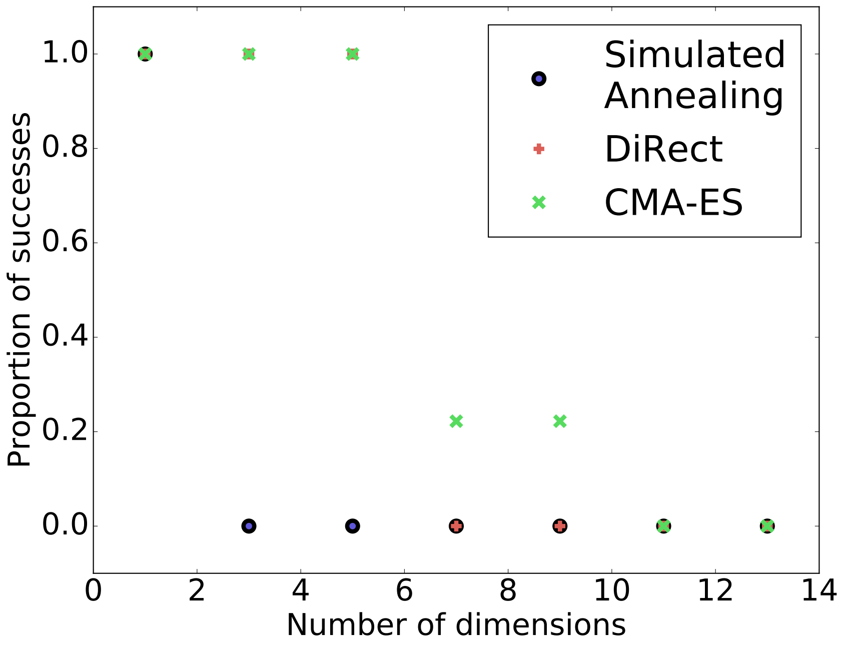


Figure 5.5: Evaluation of the performance the optimisation algorithms on the decision tree benchmark. Each dimension was evaluated over 10 run.

Luckily, in most contexts, there is some structure that can be exploited to help the optimisation converge. The benchmarks are a good example of this, while the numerical optimisers failed to find good performance values, there were closed form expressions of the optimum points. In the next section, I discuss some generic techniques which can help deal with large parameter spaces.

5.3 Decompositions

The previous section has shown that we cannot rely on off-the-shelf optimisers to find good configurations in complex configuration spaces. In this section, I discuss the use of decompositions methods to divide these large spaces into manageable chunks. Decompositions methods are generic techniques often used in numerical optimisation to tackle large dimensional spaces. On a high level, they exploit the independence of the configuration parameters to optimise them separately. This section reviews two known decomposition techniques – block decompositions (Section 5.3.1) and primal decompositions (Section 5.3.2). In both contexts, I show how BOAT's optimisation scheduling abstraction facilitates their implementation.

Note that I only cover aspects of decomposition methods which I believe are simple and generic enough to be useful to a developer. Decomposition methods have received significant research interest and many extensive techniques have been derived. In particular, I do not cover *dual* decompositions, which can be used on optimisation problems with constraints. Although they are useful, it is more complex to derive dual decompositions than their primal counterpart. As a result I do not use them in my case studies. I refer the reader to Boyd et al. [BXMM07] for an introduction to decomposition methods, and to Conejo et al. [CCMGB06] for a full treatment.

This section assumes we are given a configuration space and a structured probabilistic model and the goal is to maximise the expected performance predicted by the probabilistic model. Note that this is different from maximising an acquisition function, as is usually done at the numerical optimisation stage of a Bayesian optimisation. Chapter 6 will discuss how the techniques discussed here can be applied to Bayesian optimisations.

5.3.1 Block decompositions

This subsection introduces the overall concept that we would like to exploit: the independence of parameters. We want to optimise independent regions of the parameter space individually so that the dimension of each optimisation is small enough to be tackled by an off-the-shelf numerical optimiser. I discuss a optimisation problem where this is the case, the objective function is a sum of independent functions which can each be optimised individually. The next subsection will discuss more complex optimisation problems with a more subtle notion of independence.

Consider optimising the performance of the following function, originally discussed in Section 4.2.4:

```
size_t sort_and_hash(vector<int> a){
  sort(a);
  return hash(a);
}
```

Here, both `sort` and `hash` have their own parameters. If this were in the context of a structured Bayesian optimisation and we had to minimise the average runtime of `sort_and_hash`, we would most likely model the performance of `sort` and `hash` individually. This means that the time modelled could be written as:

$$\begin{aligned} model_{sah}(\mathbf{sort\_param}, \mathbf{hash\_param}) =& model_{sort}(\mathbf{sort\_param}) + \\ & model_{hash}(\mathbf{hash\_param}) \end{aligned}$$

The simplest way of performing the numerical optimisation of this parameter space would be to optimise all parameters together. The objective function would be the expected runtime of the corresponding implementation of `sort_and_hash`. If, however, the combined parameter spaces of `sort` and `hash` were too large to be tackled at once, we would have to extract some structure to help the optimisation converge.

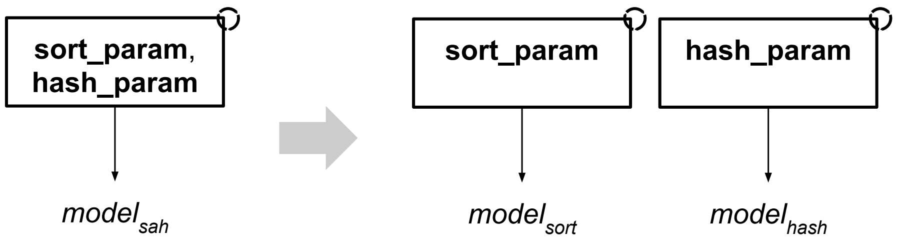


Figure 5.6: Example of block decomposition.

Here, we can do so by optimising the parameters of `sort` and `hash` individually using their respective models. We know that doing so will lead to the correct result because the performance of `sort_hash` grows monotonically with the individual performances of `sort` and `hash`. Figure 5.6 diagrammatically shows the splitting of the optimisation into two.

The parameters in this example are completely unrelated, which allows for us to optimise them independently. Most of the time, however, parameters are not unrelated but rather loosely related. I show in the next subsection how to tackle such parameter spaces.

5.3.2 Hierarchical decomposition

Often the parameter space that we wish to optimise is hierarchical. That is, some higher level parameters either have an impact on the existence of lower parameters, or on the way that they should be optimised. This subsection presents hierarchical decompositions and shows how they can be applied to both synthetic benchmarks introduced in the previous section.

The type of decomposition discussed here are also called *primal* decomposition, meaning they optimise parameters directly. On the other hand, *dual* decompositions optimise an implicit price of the parameters.

Decision tree benchmark Consider the decision tree optimisation problem from the previous section. The goal was to make a decision tree reproduce its inputs. Given a branch b, the value b_{val} of the branch has an impact on how inputs propagate to its children sub-trees. Hence, the optimal value of the children nodes will be dependent on the value b_{val}. However, for a given value of b_{val}, there is an notion of independence between each of the two children sub-trees. This is because each has a fixed input distribution that is not impacted by the specific values in the other sub-tree. Therefore, if the value of the branch was fixed, we would be able to block decompose the optimisation of both children sub-trees.

Once again, if we had to perform the numerical optimisation of this decision tree parameter space, the first step would be to apply an off-the-shelf optimiser, just like we did in the benchmark. If this did not converge, however, we would like to exploit some of the

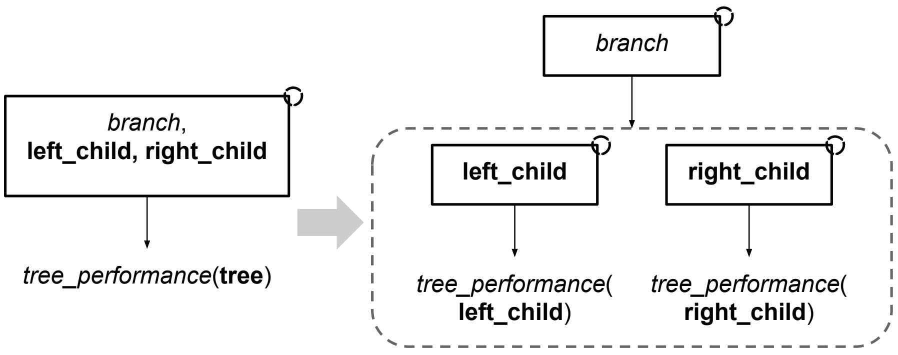


Figure 5.7: Primal decomposition of a decision tree.

```cpp
double optimise_branch_node(SortNode& branch_node,
                            InputDistribution distrib){
  SortQuery& query = branch_node.query.deref();
  SortNode& left = branch_node.left.deref();
  SortNode& right = branch_node.right.deref();

  NLOpt<> opt(query.scan_length, query.threshold);
  opt.set_utility_function([&](){
    SplitDistribution split = distrib.split_on(query);
    double left_perf = optimise_node(left, split.left_distrib);
    double right_perf = optimise_node(right, split.right_distrib);
    return weighted_average(left_perf, right_perf, split.proportion_right);
  });
  return opt.run_optimisation();
}
```

Listing 5.5: Primal decomposition to optimise a branch node and its children in the sort case study.

structure inherent to this hierarchy. Unlike in the previous subsection, it is not possible to decompose the parameter space directly. Each of the components of the tree are somewhat related.

We can exploit this structure in the following way: we perform an optimisation of the parameters of the branch node, and set as utility a procedure which performs two sub-optimisations, optimising the children sub-trees individually. The utility returned by that procedure is the sum of the performances of the children leaves l_{perf}. I show this procedure in Figure 5.7. The advantage of this method is that we are able to exploit this independence between the sub-trees.

Sort case study decision tree example. BOAT's optimisation scheduling abstraction makes it easy to implement this technique. Listing 5.5 shows the implementation of this strategy in the context of the sort case study, which also optimises a decision tree. Recall that inputs of the trees are arrays and the performance of the tree is the average time to sort the arrays, including the querying cost incurred at branches. The optimisation

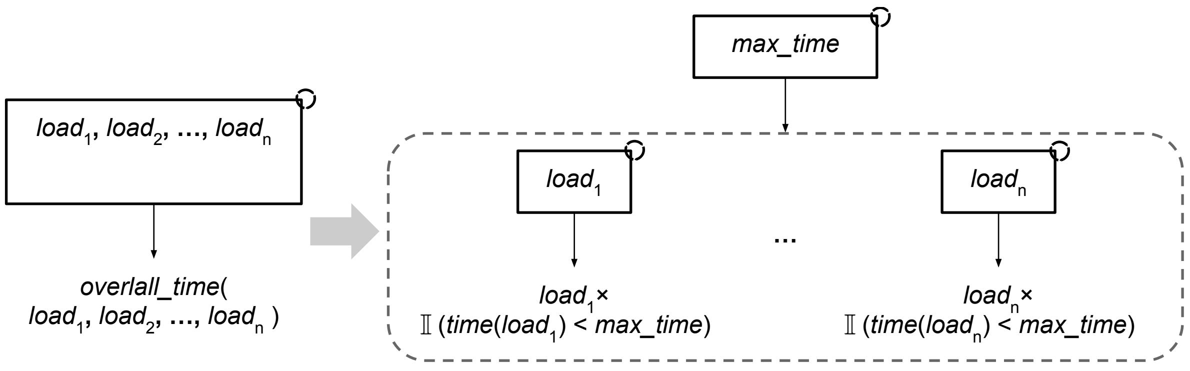

$load_1, load_2, ..., load_n$
$overlall\_time(load_1, load_2, ..., load_n)$
max_time
$load_1$
$load_1 \times \mathbb{I}(time(load_1) < max\_time)$
...
$load_n$
$load_n \times \mathbb{I}(time(load_n) < max\_time)$

Figure 5.8: Primal decomposition of the load balancing optimisation problem.

presented here optimises the parameters of a branch's query and its two sub-trees. Sort's decision tree's parameter space was defined in Section 3.3.2. The optimisation shown here would be used in the numerical optimisation stage of the Bayesian optimisation.

More specifically, the function `optimise_branch_node` takes as input a node that is a branch – the `is_leaf` parameter was already set to `false` – and a distribution of input arrays. The goal is to minimise the average time to sort the arrays in the distribution. It must optimise the configuration parameters of the query of the branch and of both sub-trees.

It does so by performing an optimisation of the query parameters, which as part its objective function performs two sub-optimisations on each of the sub-trees. On line `14`, when `run_optimisation()` is executed, the value returned is the one of the best achieved objective function value. Therefore, the optimisation returns the performance of its optimised parameters. That is, the average sort time of the input distribution in the optimised configuration.

Listing 5.5 does not show the other higher-level decompositions that are also used in the sort case study. The function `optimise_node`, used on a higher level, optimises the `is_leaf` parameter only. It performs two iterations. In the first one, it assigns `true` to `is_leaf` and delegates the optimisation of the corresponding `SortLeaf` parameters. In the second one it assigns `false` and delegates the assignment of the remaining configuration parameters to `optimise_branch_node`.

Load balance benchmark Sometimes, to perform a decomposition and exploit the independence between parameters, it is necessary to introduce a new auxiliary parameter. Consider, for example, the load balance synthetic benchmark of the previous section. Parameters represent the amount of work allocated to each worker. All parameters are related and hence no hierarchy can be directly exploited.

To remedy this, we can introduce a new parameter *max_time* which conceptually represents the maximum amount of time a worker is allowed to work for. Then, we can optimise

```
void optimise_single_param(RangeParameter<double>& load,
                           double worker_speed, double max_time){
  // Find the greatest load which does not exceed max_time
  NLOpt<> opt(load);
  opt.set_utility_function([&](){
    return load.value() * (worker_speed < max_time ? 1.0 : 0.0);
  });
  opt.set_maximising();
  opt.run_optimisation();
}

void optimise_parameters(std::vector<RangeParameter<double>>& load,
                         std::vector<double>& speeds){
  // Create the auxiliary parameter
  RangeParameter<double> max_time;
  NLOpt<> opt(max_time);
  opt.set_utility_function([&](){
    // Optimise each parameter individually
    for(int i=0; i<load.size(); i++){
      optimise_single_param(load[i], speeds[i], max_time.value());
    }
    // Measure the objective function
    double norm = 0.0, max = 0.0;
    for(int i=0; i<load.size(); i++){
      max = std::max(max, load[i].value() * speeds[i]);
      norm += load[i].value() ;
    }
    return max / norm;
  });
  opt.run_optimisation();
}
```

Listing 5.6: Implementation of the decomposition of the load balancing benchmark.

each individual parameter separately to maximise the amount of work they perform in the allocated time. Figure 5.8 shows the decomposition diagrammatically and Listing 5.6 shows its implementation in BOAT. Due to the benchmark's simplicity – the amount of work is continuous and normalised across workers – this implementation should find an optimal value on the first iteration. However, this approach would also be applicable to more complex settings. For example, in the neural network scheduling case study, I also use it. In that case, the optimisation problem is less well behaved. The units of work are discrete and the time needed by a worker is not linear with the quantity of work allocated to it. However, the decomposition approach converges quickly.

Limitations. Decompositions come at a cost. The time complexity of the optimisation is exponential with the depth of the configuration space. For example, for a decision tree of depth d, if we perform k iterations per sub-optimisation, the complexity will be $\mathcal{O}(k^d)$. Section 5.5 will tackle this by using techniques inspired by reinforcement learning. Before

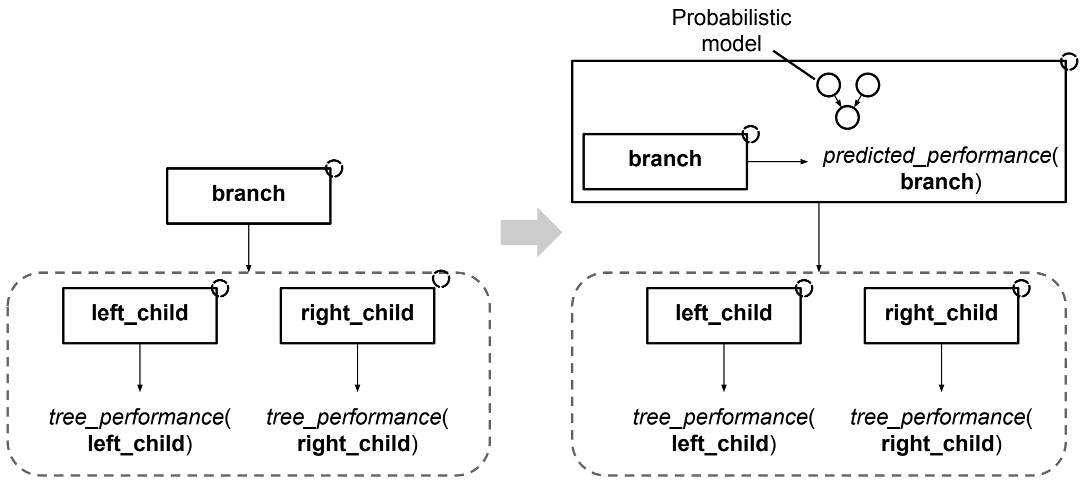

branch
left_child
right_child
tree_performance(left_child)
tree_performance(right_child)
Probabilistic model
branch
predicted_performance(branch)
left_child
right_child
tree_performance(left_child)
tree_performance(right_child)

Figure 5.9: Bayesian optimisation within a decomposition.

this, the next section introduces how to leverage the Bayesian optimisation algorithm within the numerical optimisation stage, in order to reduce the number of iterations needed by the optimisation. This will come at the cost of a higher overhead per iteration.

5.4 Using Bayesian Optimisation for decompositions

This section introduces the use of Bayesian optimisation within the numerical optimisation stage. In general, Bayesian optimisation should be used in the context of the optimisation of expensive functions. At first, the overall function that we are trying to optimise – evaluating the performance predicted by a model– is not expensive. Hence, it is not initially appropriate. To perform this, however, we use some optimisations which have as objective function other sub-optimisations, as introduced in the previous section. This can be an expensive objective function, in which case Bayesian optimisation is a sensible choice. In the remainder of this section, I will use the optimisation of a decision tree as a running example.

The simplest way in which we can consider utilising the Bayesian optimisation methodology is to use them for the optimisation of branches, which always have sub-optimisations in their objective functions. The advantage of this method is that it takes fewer iterations to converge. The disadvantage is the higher overhead per iteration, due to the numerical optimisation stage of the Bayesian optimisation, which slows down the overall procedure.

Figure 5.9 shows the schedule of replacing each numerical optimisation with a Bayesian optimisation. Note that we still use an off-the-shelf numerical optimisation to find the values predicted as best by the model.

Improving the model used by Bayesian optimisation As argued in this dissertation, when using Bayesian optimisation, it is possible to add structure to the probabilistic

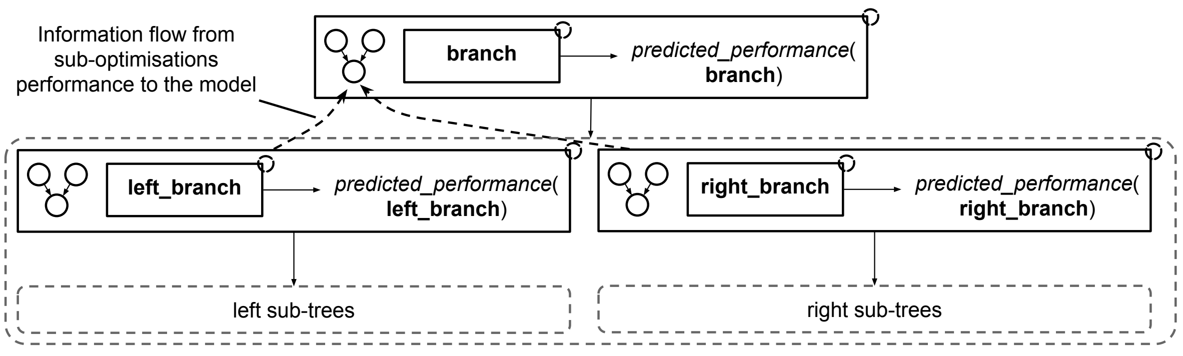

Information flow from sub-optimisations performance to the model
branch
predicted_performance(branch)
left_branch
predicted_performance(left_branch)
right_branch
predicted_performance(right_branch)
left sub-trees
right sub-trees

Figure 5.10: Structured Bayesian optimisation within a decomposition. Also shows the recursive nature of the process, with both sub-trees also performing a Bayesian optimisation.

model so that it converges in fewer iterations. Ideally, each optimisation would only require a few iterations for the model to capture the response surface of the objective function.

For example, consider the optimisation of a branch in the sort case study. Each branch has two parameters. The `scan_length` parameter specifies the length for which an input array is scanned, at a computational cost. If we were to apply traditional Bayesian optimisation to each branch decomposition, we would use a Gaussian process to model the performance of a branch's parameters. To improve on this, we can model the cost of the branch's query, and the performance of the sub-trees independently. This will expose to the model the trade-off between performing a more accurate decision at a higher cost.

Further structure can be obtained by noticing that a sub-tree's performance is uniquely dependent on the input distribution given to it. The values of the branch parameters have no direct impact on the performance of the sub-trees. Hence, we can make the model predict the performance of each sub-tree as a function of some features of their input distribution. When predicting the performance of a branch's parameter values, we first split the input distribution using these values. Then, we return the combination of the prediction of both sub-trees performance and the branch's cost. Figure 5.10 shows the resulting optimisation. The model observes the performances of each sub-tree.

Even further, we can inspect the features of these input distributions and give some knowledge to the model about the probable impact of these features by using a semi-parametric model, as presented in Section 4.3. For example, recall that in the sort case study, `block_size` is the parameter optimised at the decision trees leaves. To predict a distribution of array's performance, I compute the spread of optimal `block_size` parameter for the arrays. I place a penalty on distributions that have a large spread as they are less likely to perform well.

This exemplifies how easy it is to add structure to a Bayesian optimisation by adapting the model. As a special case, it is easy to force the prior of the model to be a known heuristic.

For example, if we were optimising classification decision trees, often used for supervised learning, we could set the prior of the model to optimise information gain [Qui86] or the Gini index, which are heuristics traditionally used in this context [Mur12]. This would mean that setting our Bayesian optimisations to perform exactly one iteration – and hence no learning – would render the same algorithm as these traditional approaches. Performing more iterations would likely yield decision trees of higher performance, albeit at an exponential cost.

The complexity of this approach remains high. The next section introduces an approach based on reinforcement learning, that is designed to improve this complexity.

5.5 A reinforcement learning inspired approach

In the previous sections, we saw methods to optimise a hierarchical parameter space which had a complexity exponential in the depth of the hierarchy. In a way, a lot of the work performed throughout these optimisations is redundant. For example, consider the optimisation of the branches at depth 1 of a decision tree using Bayesian optimisation, as described in the previous section. With each iteration of the optimisation of the root branch, they will themselves perform an entire Bayesian optimisation. These optimisations will iteratively learn the performance that can be expected out of their sub-trees. At the end of each iteration, the information they learnt is lost, although it would have been useful for subsequent iterations.

This section introduces a method based on reinforcement learning to take advantage of previous iterations' observations. I first review the general reinforcement learning approach (Section 5.5.1) and then discuss how it can be leveraged in our context (Section 5.5.2).

5.5.1 Reinforcement Learning

Reinforcement learning is an area of machine learning concerned with making actions in an environment, in order to maximise an expected reward. A reinforcement learning problem usually includes an agent and an environment. At iteration t the agent:

- Receives a state s_t the agent is in.
- Receives a scalar reward r_t.
- Selects an actions a_t from a set A of possible actions.

The environment:

- Receives action a_t.
- Emits the agent's state s_{t+1}.
- Emits a scalar reward r_{t+1}.

The role of the agent is to maximise the expected future rewards. Usually, the assumption is made that rewards are discounted by a factor $\gamma < 1$ per time step. The future discounted return at time t is $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where the game terminates at time step T.

Many algorithms have been proposed to tackle this problem. A number of them work by learning a *Q-function* which estimates the performance an agent should expect out of performing an action in a given state. It is useful to define the optimal Q-function $Q^*(s, a) = \max_\pi \mathbb{E}[R_t \mid s_t = s, a_t = a, \pi]$ where π is a policy mapping states to actions. Given, Q^*, an agent can pick optimal actions by always selecting the one predicted as best: $a_t = \arg\max_{a'} Q^*(s_t, a')$. The role of the learning will be to make a probabilistic model Q iteratively converge towards Q^*.

The $Q - learning$ algorithm [WD92] achieves this in the following way. The agent is made to run through the environment many times. Every time step, the agent either picks the best action predicted by the model $a_t = \arg\max_{a'} Q(s_t, a')$ or a random action for exploration.

Upon receiving the next state s_{t+1} and reward r_{t+1}, we estimate the performance y_t of the action. If s_{t+1} is a terminal state, meaning the environment terminates, then the performance is simply the last received reward: $y_t = r_{t+1}$. Otherwise, we use the Q-function to get an approximation of the performance of that state:

$$y_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a').$$

Finally, we use this performance as training data. The pair $((s_t, a_t), y_t)$ is fed to the model.

Initially, the training data will be of poor quality as it will be generated using the untrained model. However, as the model converges towards Q^*, the data will become of better quality. Because of this initial low data quality, the model must be able to "forget" old observations that were inaccurate. A good candidate for this task are neural networks which learn a fixed set of parameters. Hence, as more data is exposed to a neural network, the parameters can be updated in a way that no longer reflects the initial measurements. This constraint makes it difficult to use other models for supervised learning, such as Gaussian processes, as they will not forget old data.

Relationship between reinforcement learning and Bayesian optimisation. As noted by Ghahramani [Gha15], the problems tackled by Bayesian optimisation can be seen as a subclass of reinforcement learning problems in which actions (or evaluated configurations) do not affect the state of the system (the objective function). A Bayesian optimisation problem can be seen as a reinforcement learning problem with a single multi-dimensional action.

Furthermore, the approach used by Q-learning also bears similarity to the Bayesian optimisation approach. In both cases, a probabilistic model simulating the environment is iteratively learnt. Hence, if an environment only had a single multi-dimensional action, then applying the Q-learning algorithm would have the exact same procedure as applying Bayesian optimisation. Both would learn a probabilistic model mapping the action, or configuration space, to the only reward or objective function. The way in which Q-learning builds upon Bayesian optimisation is that, in the presence of multiple subsequent actions, it leverages its own model after each action to estimate how good that action was.

In the nested Bayesian optimisation approach described in the previous section, in order to evaluate the quality of a parameter value, we had to optimise the entirety of the remaining parameters. This resulted in an exponential computational complexity. In the next subsection, I show how to adapt this hierarchical decompositions with Bayesian optimisations approach to instead look like Q-learning.

5.5.2 Applying reinforcement learning techniques to the numerical optimisations

This subsection discusses the transformation of a hierarchical Bayesian optimisation, as discussed in the previous section, into a reinforcement learning-like optimisation. In both cases we have a model predicting a parameter's performance. In the Bayesian optimisation case, the data used by the model is based on the best performance achieved by the sub-optimisations. In the Q-learning algorithm case, the data is generated by the model's estimate of the bet possible performance at the next action.

Although the method presented here is not an actual reinforcement learning algorithm, it shares some of its intuition. The following mapping shows how they relate:

- **Actions** are set of parameters. For example, when I use this method to optimise a decision tree, a_0 is the value of the root node, a_1 is the values of the nodes at depth 1 and so on.
- **State**. The state is the values of all previously executed actions and their associated consequences. In the case of decision trees, the state after the first action is both the value of the root node and the resulting distributions that go to the left and right sub-trees of the root node.
- **Reward**. All actions yield a reward of 0 apart from the last one. The final action has for reward the objective function of the optimisation problem.

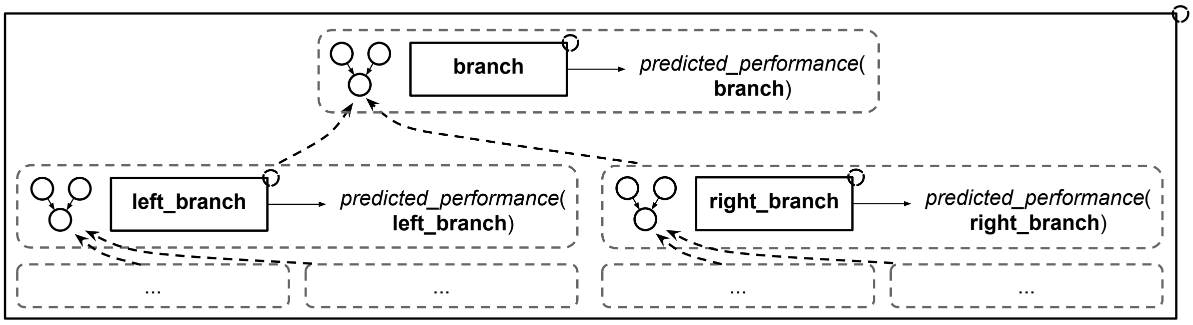

branch
predicted_performance(branch)
left_branch
predicted_performance(left_branch)
right_branch
predicted_performance(right_branch)
...
...
...
...

Figure 5.11: Reinforcement learning based approach. A model's data is generated by the predictions of the models of the lower level.

- **Model**. Instead of a simple neural network, I will use structured probabilistic models. To predict the quality of an action, these will take as input the action itself as well as all previously executed actions.

I answer two questions to make reinforcement learning applicable to our context:

- How can we leverage a possibly inaccurate estimate of the performance with our models? Neural networks, often used for reinforcement learning, have the ability to "forget" old measurements. How can the same behaviour be achieved with our models based on full Bayesian inference?
- How can we get such an estimate by only looking at the next action?

The first question is tackled by leveraging the notion of *noise* that was always used in the probabilistic models presented in Chapter 4. Given an observation, our models are always able to take as input the noise of the observation. This is true of models using Gaussian processes and treed Gaussian processes. The measurements have a Gaussian likelihood and a noise variable can be passed representing its variance.

Therefore, one approach would be to associate a large noise to each measurement. This would be wasteful, however, as we would like to discriminate between observations which are performed early on – and are likely poor – and higher quality ones performed later on. Hence, we need to have an estimate of the inaccuracy of each measurement.

The second question is hence two-fold: how can we get this estimate and its variance by only looking at the next action. Getting an estimate can be done in a similar way as is done in Q-learning: we optimise the subsequent action and measure the predicted performance of these optimised action.

For example, with a decision tree, to estimate the performance of the parameters of a branch at depth d, we optimise the parameters of the top branches of both its sub-tree, at depth $d+1$. The optimisation is done using each branch's respective probabilistic model.

Then, we use the performance predicted by the models for these optimised branches as a proxy for how well we can hope to optimise the rest of the sub-tree. Figure 5.11 shows this procedure diagrammatically. Unlike in the previous section, we are no longer optimising a branch's sub-trees as part of the branch iterative optimisation. Instead, with each iteration of a global optimisation, we go down the entire tree optimising each branch according to its own model.

We also need an estimate of the variance. I achieve this by generating Thompson samples of the branch and measuring the variance in predicted performance. Recall from Section 2.2.2 that Thompson sampling involves sampling a model from the distribution of models. Then, we find the optimal configuration for that model sample [Tho33]. Each time I optimise a branch, I sample multiple independent models and optimise the branch using each model independently. The resulting performance of these optimisations allow me to estimate the variance in the distribution of the optimal performance for that sub-tree. Sampling can be done in Probabilistic-C++ using the method described in Section 4.6. Hence, this approach allows us to compute both the average predicted performance as well as the noise.

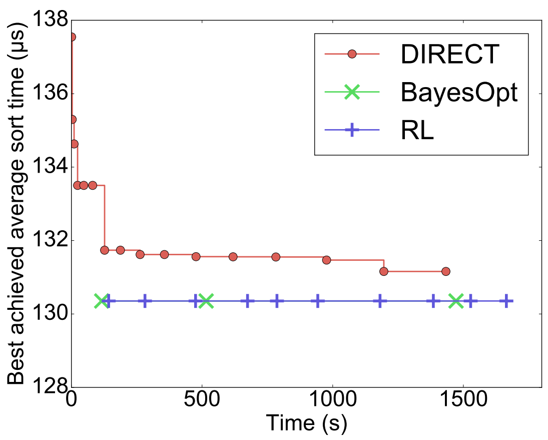
Note that this reinforcement learning approach is the last and most complex method I propose to help the numerical optimisation converge. For this reason, it should only be used in a context where applying recursive Bayesian optimisation, as described in the previous section, leads to a too long convergence time. I only use this approach in the context of the sort case study to optimise the decision trees. In the next section, I compare the performance of the different decomposition methodologies presented in this chapter on an optimisation problem from the sort case study.

5.6 Evaluation

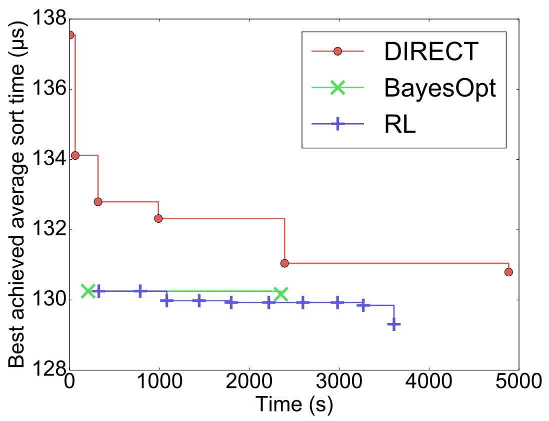
In this section I compare the empirical performance of the different decomposition techniques introduced in this chapter. I do so by evaluating their performance on the numerical optimisation stage of the sort case study. I load a model that was stored after performing an entire structured Bayesian optimisation. I also load the corresponding array distribution. Then, the optimisation problem is: optimise the decision tree parameter space to minimise the predicted average sort time.

I evaluated three procedures. First, a simple hierarchical decomposition, as described in Section 5.3.2, which uses the DIRECT algorithm for each of its optimisations.

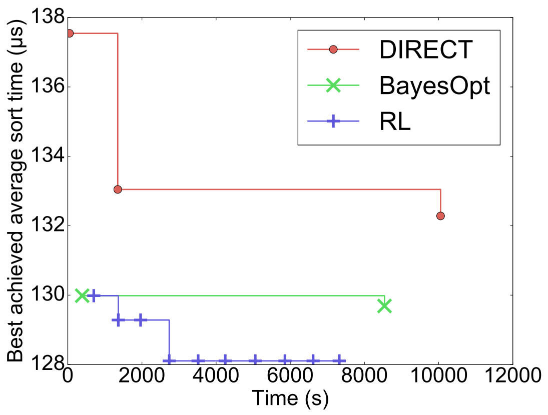
Second, a hierarchical decomposition which uses a structured Bayesian optimisation to optimise each of its branches. The model, implemented in Probabilistic-C++, takes as input an input distribution, a `SortQuery` object and the maximum depth of the node. It computes the two distributions that will go in each sub-tree, along with some of their



(a) Tree depth of 4. DIRECT executed for up to 80 iterations per optimisation. Bayesian optimisation executed for up to 3 iterations per optimisation.



(b) Tree depth of 5. DIRECT executed for up to 30 iterations per optimisation. Bayesian optimisation executed for up to 2 iterations per optimisation.



(c) Tree depth of 6. DIRECT executed for up to 15 iterations per optimisation. Bayesian optimisation executed for up to 2 iterations per optimisation.

Figure 5.12: Comparison of the best achieved average sort time by the different optimisation approaches.

summary statistics. These statistics include the best and worst performance that can be achieved on each of the distributions. Finally, it uses a treed Gaussian process to predict the performance that will be achieved by each sub-tree within this possible spectrum.

Third, a reinforcement learning based decomposition which uses the same model as the structured Bayesian optimisation based one. Instead of using one instance of the model per branch, as described in the previous section, I share the same model across all branches. Each time a query is being optimised, the optimisation is performed three times. Once to maximise the average performance predicted by the model, and the other two by using samples from the model. This generates a tuple *(input distribution, maximum depth, performance estimate, variance estimate)*. At the end of each iteration, the tuples generated by all branches are fed into the model.

Iterations are alternatively performed in *exploitation* and *exploration* mode. In exploitation mode, once the three optimisations have been performed on a query, the value that is kept is the one that was optimal for the average model prediction. In exploration mode, the value that is kept is the one that yielded the most optimistic predicted performance.

Figure 5.12 shows the results. I optimised trees of depth 4, 5 and 6. For the two hierarchical decomposition approach, I repeated the optimisations multiple times, each time allowing a different number of iterations per optimisation. I report the time that each optimisation took, along with the best achieved result. In the reinforcement learning based approach, I ran the optimisation for ten iterations, and the different data points show the performance achieved so far. This means that the results are obtained incrementally, whereas for the primal decomposition approaches, the optimisation had to be run from scratch. The runtimes were obtained by running the optimisations on an `m4.xlarge` EC2 instance.

There are a few interesting points to note. First, the reinforcement learning approach shows a slight improvement over the Bayesian optimisation approach, which itself performs significantly better than simply using the DIRECT algorithm. In particular, the Bayesian optimisation always performs decently on its first iteration, suggesting that the model I designed to predict a branch's performance has a good prior.

Second, compare the first point of the Bayesian optimisation and the reinforcement learning inspired approach on each graph. Both consistently perform equally well but the reinforcement learning approach takes longer. The reason is that both approaches use the same model. Hence, in the first iteration, when no data has been given to the model, they both optimise the branches in the same way. However, the reinforcement learning approach also performs Thompson samples, hence the extra computational cost.

Third, the two recursive decomposition approaches do not scale with the depth of the tree. The performance of the DIRECT based decomposition worsens from increasing the depth of the tree, even though the representative power of the configuration space is strictly greater. In the Bayesian optimisation case, the cost is such that we can only run a maximum of two iterations per optimisation on trees of depth 5 and 6. In contrast, the time needed to do a fixed number of iterations with the reinforcement learning approach grows roughly linearly with the number of nodes in the tree.

5.7 Summary

In this chapter, I discussed two related concepts. First, I presented BOAT's optimisation scheduling abstraction which provides a structured framework to build complex optimisations. In particular, it can be used to implement nested optimisations, a frequently occurring pattern.

Second, I have discussed the difficulty of performing the numerical optimisation stage of Bayesian optimisations. I have shown how off-the-shelf optimisers can be inadequate for the task of optimising domains with more than five parameters. I therefore introduced the use of decompositions as part of this numerical optimisation stage. Decompositions exploit some of the independence of large optimisation problems to divide them into manageable chunks.

I have presented two generic techniques to improve the performance of decomposed optimisations. First, using Bayesian optimisation to perform the decompositions. This reduces the number of iterations, at the cost of a higher overhead per iteration. Second, by extending this first approach in a manner similar to reinforcement learning. In this case, the predictions made by a probabilistic model are used as an estimate of the true performance. I empirically compared the performance of these different approaches on a numerical optimisation problem from the sort case study.

CHAPTER 6

Designing a bespoke auto-tuner

This chapter puts together the techniques from the previous three chapters to design a structured Bayesian optimisation. It makes the following contributions:

- I present how a structured Bayesian optimisation can be implemented in BOAT (Section 6.1).
- I show how to measure the expected improvement (EI) acquisition function of a structured model (Section 6.2). EI is a general acquisition function which naturally balances exploration and exploitation, and I use it in my case studies.
- I show how to find a configuration which maximises the expected improvement via Thompson samples (Section 6.3). In particular, it allows for the use of decomposition methods presented in Chapter 5. The approach is well suited to BOAT's optimisation abstraction.
- I present a method to exploit *cheap experiments*. Having a structured model means we can perform inference on experimental results other than the expensive objective function, allowing the model to converge faster towards the objective function. I propose a new approach, based on BOAT's optimisation scheduling abstraction, which makes the scheduling of experiments clear and intuitive to a user (Section 6.4).
- I conclude with the general methodology which should be followed when building a bespoke auto-tuner with BOAT (Section 6.5).

Note that this chapter contains no evaluation. Most sections present generic techniques for structured Bayesian optimisation and hence will be implicitly evaluated through the case studies in Chapter 7. Unfortunately, none of the three case studies that I considered for this dissertation were amenable to cheap experiments, and hence evaluating the approach presented in Section 6.4 is out of the scope of this dissertation. I however discuss a possible extension of the neural network case study which would benefit from this approach.

6.1 Declaring a structured Bayesian optimisation in BOAT

This section goes through the process of declaring a structured Bayesian optimisation in BOAT. As an illustrative example, Listing 6.1 shows the implementation of the garbage collection (GC) case study's Bayesian optimisation. Recall from Section 3.1.1 that in the GC case study I optimise the value of three JVM flags to minimise the 99th percentile latency of a database.

The first step in building a structured Bayesian optimisation is defining the configuration space being optimised. In the GC case, it is a `GCFlags` object containing three `RangeParameter`s for the three flags being optimised.

Then, we declare the probabilistic model used by the optimisation. In the GC case study, I use three independent model classes: `GCRateModel` predicting the rate of GCs, `GCDurationModel` predicting their duration and `LatencyModel` predicting the 99th percentile latency of the database. I build a `ProbEngine` for each class.

A structured Bayesian can then be declared via the BOAT's `BayesOpt` class. It is templated on the type of measurements which will be returned by the objective function. In our case, this is a `GCResult` class which stores the rate and duration of a GC, and the corresponding latency observed in an experiment.

In order to be run, three functions must be given to the `BayesOpt` object:

- The *suboptimisation* function is responsible for assigning values to the parameters. This is where the numerical optimisation will be executed. In the GC case study, I use the expected improvement (EI) acquisition function, which maximises the expected gain over the *incumbent*: the best result obtained so far. The function `optimise_expected_improvement` performs a numerical optimisation. It leaves the parameters of `gc_flags` assigned with values which maximise the EI, given the incumbent and the different parts of the model.

- The *objective* function executes the experiment and returns the result. For the GC case study, this is a `GCResult` object.

- The *learning* function takes as input the result measured this iteration and uses it to perform inference on the model. Each model takes as input the values used for the flags as well as their relevant results.

```cpp
//Declare the configuration space
GCFlags gc_flags;

// Declare the model
ProbEngine<GCRateModel> rate_model;
ProbEngine<GCDurationModel> duration_model;
ProbEngine<LatencyModel> latency_model;

// Declare the optimisation
BayesOpt<GCResult> opt;
opt.set_suboptimisation_function(
  [&](){
    double incumbent = opt.get_best_utility();
    optimise_expected_improvement(gc_flags, incumbent, rate_model,
                                  duration_model, latency_model);
  });

opt.set_objective_function(
  [&](){return evaluate(gc_flags);});

opt.set_learning_function(
  [&](const GCResult& result){
    rate_model.observe(gc_flags.ygs.value(), gc_flags.sr.value(),
                       gc_flags.mtt.value(), result.rate);
    duration_model.observe(gc_flags.ygs.value(), gc_flags.sr.value(),
                           gc_flags.mtt.value(), result.duration);
    latency_model.observe(gc_flags.ygs.value(), gc_flags.sr.value(),
                          gc_flags.mtt.value(), result.rate,
                          result.duration, result.latency);
  });
// Set the optimisation properties
opt.set_max_num_iterations(10);
opt.set_minimising();

// Run the optimisation
opt.run_optimisation();
```

Listing 6.1: Implementation of the structured Bayesian optimisation in the garbage collection case study.

Then, we must set some of the optimisation properties, such as the number of iterations. Finally, the optimisation can be run. After it has returned, the parameters in `gc_flags` will be left assigned the values which generated the best latency.

Choice of acquisition function. By default in my case studies, I use the expected improvement acquisition function. I have found it generally performs well. Furthermore, it is simple to estimate in the context of structured probabilistic models. Future work could investigate whether more elaborate acquisition functions, such as predictive entropy search [HLHG14], would yield better convergence.

```cpp
// The sample function of a semi-parametric model
double GCRateModel::sample(int ygs, int sr, int mtt){
  return gp.sample({ygs, sr, mtt}) + parametric(ygs, sr);
}
/*-----------------------------------------------------------------------*/

double expected_improvement(
    GCFlags& gc_flags, double incumbent,
    ProbEngine<GCRateModel>& rate_model,
    ProbEngine<GCDurationModel>& duration_model,
    ProbEngine<LatencyModel>& latency_model){
  double ei = average([&](){
    double ygs = gc_flags.ygs.value();
    double sr = gc_flags.sr.value();
    double mtt = gc_flags.mtt.value();
    double rate = rate_model.sample(ygs, sr, mtt);
    double duration = duration_model.sample(ygs, sr, mtt);
    double latency = latency_model.sample(ygs, sr, mtt, rate, duration);
    return max(incumbent - latency, 0.0);
  });
  return ei;
}
```

Listing 6.2: Measuring the expected improvement of a GC configuration.

This section has demonstrated how to declare a Bayesian optimisation in BOAT. In order to be complete, the implementation must also specify how to maximise the expected improvement. I describe a method to do so in Section 6.3. The next section discusses how to measure the expected improvement.

6.2 Measuring the expected improvement

In order to select a promising configuration, Bayesian optimisation uses an acquisition function that is designed to trade-off exploration and exploitation. In my case studies, I use the expected improvement acquisition function as it performs well and is simple to estimate. This section describes how to measure the expected improvement of a configuration, given an incumbent – the best result obtained so far – and a structured probabilistic model.

Recall that the expected improvement of a configuration $\mathbf{x}$ with incumbent η and model G is

$$\alpha_{\text{EI}}(\mathbf{x} \mid \eta, G) = \int \max(0, g(\mathbf{x}) - \eta)\, p(g(\mathbf{x}) \mid G)\, dg(\mathbf{x}).$$

That is, we must compute the average value of $\max(0, g(\mathbf{x}) - \eta)$, where $g(\mathbf{x})$ is a model's prediction for $\mathbf{x}$. In Probabilistic-C++, we can compute the average of an arbitrary function of multiple models using the `average` function introduced in Section 4.2.4.2.

One issue comes from the use of Gaussian processes within models which predict as output a normal distribution rather than a single value. The solution is to sample from this output distribution. If a Gaussian process is only queried once per prediction, we can use the `GaussianProcess::sample()` or `TreedGP::sample()` member functions which do this automatically. I show this in the context of the GC case study in Listing 6.2.

One further difficulty occurs if making a prediction requires sampling from a single Gaussian process multiple times. Then, whenever a Gaussian process is sampled from, we must make sure it remembers that sample so that future samples are distributed accordingly. This can be achieved by the functions `GaussianProcess::sample_and_observe()` or `TreedGP::sample_and_observe()`. However, this will leave the corresponding models with useless, randomly generated observations that may be wrong. Hence, before executing an `average` which uses `sample_and_observe`, I always first create a temporary copies of the models and use these copies to predict the expected improvement.

Using this technique, the expected improvement of a configuration can be measured. The next section discusses how to find configurations which maximise its value.

6.3 Maximising the expected improvement

This section discusses how to find configurations with high expected improvement values. There are two issues with optimising the expected improvement directly:

- **Highly multi-modality.** The expected improvement function is highly multi modal. It returns low values at configurations which have already been observed, and high values at promising configurations. If after some iterations of the optimisation we have sampled a few points near the true optimum, then the response surface of such acquisition functions near the optimum will be highly jagged. This makes the task of the numerical optimiser difficult. In practice, I have seen the DIRECT optimisation algorithm fail to find configurations with good expected improvement on the Branin-Hoo synthetic benchmark [Jon01], which only has two dimensions. This unreliability is undesirable when designing an auto-tuner.

- **Amenability to decompositions.** If the dimensionality of the optimisation problem is large, then it may be necessary to decompose it into sub-optimisation problems in order for the numerical optimisation to converge, as presented in the previous chapter. Unfortunately, acquisition functions like the expected improvement are not easily amenable to decompositions. For example, consider the following model which is the sum of two independent models: $G(\mathbf{x}_1, \mathbf{x}_2) = G_1(\mathbf{x}_1) + G_2(\mathbf{x}_2)$. If we were

```
void optimise_expected_improvement(
  GCFlags& gc_flags, double incumbent,
  ProbEngine<GCRateModel>& rate_model,
  ProbEngine<GCDurationModel>& duration_model,
  ProbEngine<LatencyModel>& latency_model){

  SimpleOpt<> opt;
  opt.set_iteration_function([&](){
    // For each model, sample a single particle and set the non-parametric
    // model to sampling mode
    ProbEngine<GCRateModel> rate_model_sample =
        rate_model.single_particle_enginge();
    rate_model_sample.execute([](GCRateModel& m){m.gp.set_sampling();});
    ...

    // Find a configuration which maximises the sample's performance
    optimise_performance(gc_flags, rate_model_sample,
                         duration_model_sample, latency_model_sample);

    // Return the expected improvement over the entire model
    return expected_improvement(gc_flags, incumbent, rate_model,
                                duration_model, latency_model);
  });
  opt.set_maximising();
  // Generate 100 Thompson samples
  opt.set_max_num_iterations(100);
  opt.run_optimisation();
}
```

Listing 6.3: Implementation of the Thompson sampling numerical optimisation method in the GC case study.

optimising the expected value of G, then we could use:

$$\begin{aligned}\mathbf{E}_G(G(\mathbf{x}_1, \mathbf{x}_2)) &= \mathbf{E}_{G_1,G_2}(G_1(\mathbf{x}_1) + G_2(\mathbf{x}_2)) \\ &= \mathbf{E}_{G_1}(G_1(\mathbf{x}_1)) + \mathbf{E}_{G_2}(G_2(\mathbf{x}_2))\end{aligned}$$

and block decompose $\mathbf{x}_1$ and $\mathbf{x}_2$ to optimise them separately. On the other hand, if we were optimising the expected improvement over a previous incumbent η:

$$\alpha_{\text{EI}}(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{E}_{G_1,G_2}(\max(0, (G_1(\mathbf{x}_1) + G_2(\mathbf{x}_2) - \eta))$$

is not directly decomposable.

As an alternative to optimising the expected improvement directly, I propose generating many Thompson samples and selecting the one with highest expected improvement. That is, we repeat the following procedure many times. First, we sample a model from the posterior distribution of models. In the context of Probabilistic-C++, this can be

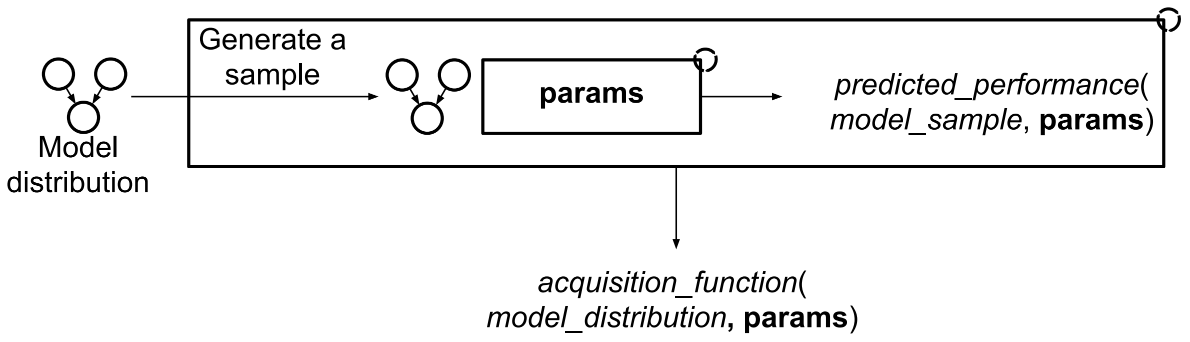

Model distribution
Generate a sample
params
predicted_performance(model_sample, params)
acquisition_function(model_distribution, params)

Figure 6.1: Numerical optimisation stage with Thompson samples.

approximated using the method described in Section 4.6. Second, we perform a numerical optimisation to find the configuration with highest performance predicted by that sample. Because we are optimising a predicted performance, the objective function is both smooth and decomposable. Third, we measure the expected improvement of that configuration. Once this procedure has be ran many times, we select the configuration which yield the highest expected improvement. This is the configuration selected by the numerical optimisation stage of the Bayesian optimisation.

I show in Listing 6.3 the implementation of this procedure in the context of the GC case study. It uses BOAT's optimisation abstraction to perform multiple independent optimisations. Each iteration the optimisation samples a model from the model distribution. It then performs a numerical optimisation to find a configuration which maximises the performance predicted by this model's sample. Finally, it returns the expected improvement of the configuration. Figure 6.1 shows this procedure diagrammatically.

This procedure is similar to the one used by portfolio acquisition functions [HBdF11, SWH$^+$14], as discussed in Section 2.2.2. They also perform multiple independent numerical optimisation and select a configuration based on a meta-criterion. However, the motivation is different. The goal here is to be able to use decompositions while also performing exploratory experiments, whereas portfolio acquisition functions are trying to select the most performant acquisition function for the problem at hand.

Interestingly, because the sampling mechanism was inspired by the UCB acquisition function [SKSK10], performing numerical optimisations on samples of a model made of a single Gaussian process is similar to optimising the UCB acquisition function with a random value for κ.

Another benefit of this approach, with high practical implication, is its computational efficiency. Evaluating the performance of a configuration using a single model sample is much cheaper than averaging over the entire distribution. Hence, it makes sense to use only one sample when performing the expensive search for good configurations. Once we have selected a few promising configurations, we select the most favoured one using the expensive expected improvement which averages over the entire distribution.

I also find that the approach presented in this section is robust. Performing multiple independent optimisations protects us from infrequent issues that may occur in the numerical optimisation, such as getting stuck in local minima. I therefore use it in all of my case studies.

Number of Thompson samples. There is a trade-off between computational overhead and accuracy in the number of Thompson samples generated. More samples will possibly lead to a better picked configuration, at the cost of a higher overhead per iteration of the Bayesian optimisation. In the garbage collection and neural network case studies, I used 100 samples. On the other hand, the sort case study suffers from a much higher cost of the numerical optimisation. Generating a single sample took approximately 15 minutes. Hence, I only used one sample, effectively performing Thompson sampling. For this reason, the sort case study required up to twenty iterations to converge, versus ten for the other two case studies.

Limitations. Recall that the approach introduced in Section 4.6 to sample a model from the model distribution is an approximation. If the model contains a Gaussian process, then the entire response surface of the sampled Gaussian process will either be lower or higher than its mean value. In some cases, this approximation may prevent the optimisation from exploring promising regions of the configuration space. For example, this could happen if the good performance of a configuration was dependent on a single Gaussian process being high at a certain point and low at another. In practice, I have not observed this being an issue.

This concludes the set of necessary techniques needed to implement a structured Bayesian optimisation. The next section discusses a more advanced technique to use cheap experiments.

6.4 Exploiting cheap experiments

Often when optimising the performance of a computer system, we can run some small experiments which give us some information about the underlying behaviour at a fraction of the cost of the objective function. For example, these can evaluate a subcomponent of the system, or evaluate the system on a reduced benchmark. In this section, I propose a method to easily leverage such experiments to make the optimisation converge.

In the most generic approach to the problem, there is a domain of possible experiments. We have to perform a joint optimisation to find a pair of experiment and configuration which tells us the most about the region of the optimum. If experiments have a range of costs, these could be included in order to maximise the rate at which we learn about the optimum's region. In general, this approach is difficult to apply. It involves measuring a metric of what is expected to be learnt about the optimum, such as the information gain.

Instead, I propose in this section an approach which is intuitive and naturally exploits BOAT's optimisation scheduling abstraction. It relies on the assumption that the cheap experiments that are available are at least an order of magnitude cheaper than the objective function. Hence, they can be executed many times per objective function evaluation, at a small increase in total cost. Unfortunately, none of my case studies had such a context and hence evaluating this procedure is outside the scope of this dissertation.

In Section 6.4.1, I describe an example problem which serves as motivation. I then describe my proposed approach (Section 6.4.2). One of the difficulties is the design of the acquisition function, which I discuss in Section 6.4.3.

6.4.1 Example problem

This subsection gives a motivational example of a setting with an experiment orders of magnitude cheaper than the objective function. In the neural network case study, I optimised the scheduling of a neural network on a distributed heterogeneous cluster. The goal was to find the schedule which minimises the time per iteration of stochastic gradient descent (SGD). Consider an optimisation problem where we have to configure both the scheduling and the architecture of a neural network. The architecture setting could include the number of layers and the number of neurons per layer. The optimisation task is to find the global configuration which, when trained on a given dataset for a day, yields the highest classification accuracy.

When considering the neural network architecture, there is a trade-off between the capability of the network – larger neural networks will eventually converge to better classification errors – and computational performance – smaller networks will be computationally cheaper and hence be able to train for more iterations within a day.

Although it is dependent on all of the architecture configuration parameters, this trade-off is completely encapsulated by the number of SGD iterations that the network will be able to perform in a day. Hence, a model predicting the final classification accuracy of a configuration would likely first estimate the number of SGD iterations that would be executed within the day from the architecture and scheduling parameters. Then, it would predict the final performance as a function of the architecture parameters and the estimated number of iterations.

In this context, I would propose using the evaluation of the time per iteration of stochastic gradient descent – which takes a few minutes to estimate accurately – as a cheap experiment. I now describe the concept of *partially experimental models* which are key to my approach.

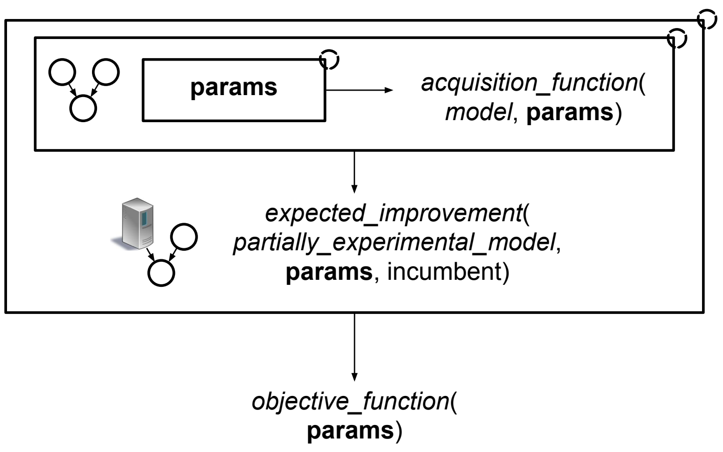

params
acquisition_function(model, params)
expected_improvement(partially_experimental_model, params, incumbent)
objective_function(params)

Figure 6.2: Bayesian sub-optimisation with cheap experiments.

6.4.2 Partially experimental models

In the context of structured Bayesian optimisation, I define a partially experimental model as a model that predicts the output of the objective function and either:

- can take as input an experimental result from a cheap experiment to make its prediction (a *discriminative* model), or,
- can perform inference on an experimental result from a cheap experiment to update its objective function prediction (a *generative* model).

When we evaluate the prediction of a partially experimental model, we first execute the cheap experiment and use the result to make an informed prediction. For example, in the neural network example, the model would be able to take as input an empirically measured time per SGD iteration to predict a configuration's performance. If it is possible to design such a model, then I propose using it for cheap experiments in the following way.

We perform two nested structured Bayesian optimisations. The high level optimisation maximises the true objective function. The sub-optimisation is analogous to the numerical optimisation stage, and maximises the expected improvement predicted by the partially experimental model. Because getting predictions from the partially experimental model is somewhat expensive, as it requires executing the cheap experiment, it cannot be optimised directly by a numerical optimiser, hence the use of a Bayesian sub-optimisation. Figure 6.2 shows the overall structure of the optimisation. Once again, this is easily expressible using BOAT's optimisation abstraction.

The next subsection will discuss the range of possibilities for the inner Bayesian optimisations acquisition function. I here list some of the benefits of this approach.

- There is an explicit balance between cheap experiments and the objective function. By setting the number of iterations for which the inner Bayesian optimisation is executed, a developer can specify how many cheap experiments can be run per evaluation of the objective function. BOAT also includes some elaborate stopping criteria for optimisation which asses the rate of improvement over past k iterations, where k is a parameter. Using such stopping criteria for the inner optimisation will initially make the optimisation explore the cheap experiment heavily when there is high uncertainty about its behaviour. In later iterations the inner optimisation will quickly converge, hence performing fewer cheap experiments.

- Cheap experiments are only evaluated on promising configurations. This guarantees that we will not be exploring low performance regions of the configuration space.

- The objective function is only evaluated on configurations which the cheap experiment was previously evaluated on. This guarantees that we never use the expensive objective function to explore properties observable in the cheap experiment.

Related work. The approach proposed here is similar in concept to the ones proposed by Swersky et al. [SSA14] and Domhan et al. [DSH15]. In these works, the goal is to tune the hyperparameters of machine learning models to maximise their performance after training. They use the measured performance after training the model for a reduced time as a cheap experiment. Both approaches develop partially experimental models, which update their predictions of the final performance in light of these measurements. However, they also rely on the fact that the cheap experiment can simply be continued to yield the expensive experiments, something that I do not assume here.

6.4.3 Acquisition functions for partially experimental models

One difficulty that arises is the following: within the inner Bayesian optimisation, which acquisition function should be used? Say, for example, that the high level Bayesian optimisation uses the expected improvement predicted by the partially experimental model as acquisition function. This means the inner Bayesian optimisation's objective function will be this expected improvement. Therefore, it needs to tackle the exploration-exploitation trade-off and find configurations which may improve on the best expected improvement found so far.

Assume that we use the method for the numerical optimisation presented in the previous section. We have a set of Thompson sampled configurations and we need to discriminate which should be selected by the inner optimisation for the cheap experiment to be performed on. Initially we can select ones with high expected improvement of the overall objective function. However, if we do this for a while, we run the risk of performing the same cheap experiment every iteration while discarding some configurations with lower expected improvement but more informative cheap experiments.

I highlight below three possible alternatives.

Expected improvement's expected improvement. Say, for example, that the best value of the objective function found so far is η, and the best configuration $\mathbf{x}_+$ found by the inner optimisation so far yielded expected improvement $\nu = \alpha_{\mathrm{EI}}(\mathbf{x}_+ \mid \eta, G)$, then the expected improvement's expected improvement can be computed as:

$$\begin{aligned}
&\alpha_{\mathrm{EI-EI}}(\mathbf{x} \mid \eta, \nu, G) \\
&\quad= \int \max\left(0, \alpha_{\mathrm{EI}}(\mathbf{x} \mid \eta, g_{ch}(\mathbf{x})) - \nu\right)\ p(g_{ch}(\mathbf{x}) \mid G)\ dg_{ch}(\mathbf{x}) \\
&\quad= \int \max\left(0, \int \max(0, g_{obj}(\mathbf{x}) - \eta) p(g_{obj}(\mathbf{x}) \mid G, g_{ch}(\mathbf{x})) dg_{obj}(\mathbf{x}) - \nu\right) \\
&\qquad p(g_{ch}(\mathbf{x}) \mid G) dg_{ch}(\mathbf{x}).
\end{aligned}$$

Where g_{ch} and g_{obj} are predictions of the model G for the cheap experiment and objective function respectively. The difficulty with this approach is that it requires performing a double integral over the probabilistic model. We must sample possible values for the outcome of the cheap experiment, and for each sample compute the expected improvement of the overall utility function that would ensue. Depending on the model, this type of double integral may be intractable to compute accurately in a reasonable time. Further, it is not directly implementable in probabilistic programming, as it requires averaging over a procedure which itself computes an average.

UCB inspired approach. As a second alternative, I propose an acquisition function inspired by the upper confidence bound acquisition function, introduced in Section 2.2.2. It assumes that we have a scalar metric for the model's standard deviation of the outcome of the cheap experiment $\sigma_{ch}(\mathbf{x})$. This may not be straightforward to design when the cheap experiment results in multiple observations $\mathbf{f}_{ch}(\mathbf{x}) = f_{ch}^{(1)}(\mathbf{x}), f_{ch}^{(2)}(\mathbf{x}), \ldots, f_{ch}^{(n)}(\mathbf{x})$, although in some cases, the root of the mean variance across predictions may be an appropriate choice:

$$\sigma_{ch}(\mathbf{x}) = \sqrt{\sum_{i=1}^{n} \left(\sigma_{ch}^{(i)}(\mathbf{x})\right)^2}.$$

In the neural network example, we could simply use the standard deviation of the expected runtime. We can then trade-off exploration and exploitation of the cheap experiment by using the following function:

$$\alpha_{\mathrm{UCB-EI}}(\mathbf{x} \mid \eta, G) = \alpha_{\mathrm{EI}}(\mathbf{x} \mid \eta, G) + \kappa_{ch}\sigma_{ch}(\mathbf{x}).$$

The first term is the expected improvement of executing the expensive objective function. This is the quantity that we are trying to maximise. It is analogous to the $\mu(\mathbf{x})$ term of the upper confidence bound acquisition function. The overall expected improvement, including the uncertainty about $f_{ch}(\mathbf{x})$, is equal to the average value of the expected improvement after observing the outcome of $f_{ch}(\mathbf{x})$. The second term favours explorations.

The variable κ_{ch} is an algorithm parameter which must be tuned to the particular setting in which the optimisation will be run. This is also true of the equivalent variable of the UCB acquisition function, but it is more important in this context as the units of $\alpha_{\text{EI}}(\mathbf{x} \mid \eta, G)$ and $\sigma_{ch}(\mathbf{x})$ may be different. Overall, this acquisition function selects configurations with high expected improvement which also have some uncertainty about the outcome of the cheap experiment.

One positive aspect of this approach is that it is applicable to contexts where we have many cheap experiments with a range of costs, and hence we are performing multiple nested Bayesian optimisations. If we had an even cheaper experiment $f_{ch2}(\mathbf{x})$, we could nest another Bayesian optimisation and maximise

$$\alpha_{\text{UCB-EI}}(\mathbf{x} \mid \eta, G) = \alpha_{\text{EI}}(\mathbf{x} \mid \eta, G) + \kappa_{ch}\sigma_{ch}(\mathbf{x}) + \kappa_{ch2}\sigma_{ch2}(\mathbf{x})$$

in the inner most one.

Thompson sampling. One third and final alternative is to discard the acquisition function and simply evaluate the cheap experiment on each Thompson sample. Although wasteful, this approach will thoroughly explore the domain at a small overhead per iteration.

Note that all three approaches described sit at different points in the trade-off between accuracy of the selected configuration and computational complexity. All three should eventually converge, but the earlier approaches will do so in fewer iterations with a high overhead per iteration. When considering which to use for a specific problem, one should consider the cost of the cheap experiment itself. If it is expensive, a higher overhead per iteration will seem favourable. Otherwise, a higher rate of iterations may help the model converge faster. This trade-off is not specific to the use of cheap experiments, it is inherent to the numerical optimisation stage of any Bayesian optimisation. I discuss this further in the next section on incrementally designing a Bayesian optimisation.

6.5 Incrementally building a bespoke auto-tuner with BOAT

In this section I describe the general approach to building a bespoke auto-tuner with BOAT. Like in other aspects of software engineering, *premature optimisation* [Knu74] is an easy pitfall to succumb to. Hence, I advocate that the correct way to build an auto-tuner is to start with a naive solution and incrementally refine it so it converges in a reasonable time.

Once a developer has defined the configuration space and the objective function of their application, they should start by implementing traditional Bayesian optimisation using a Gaussian process. This may involve only optimising a subset of the parameter space, using $\mathbf{x} \in \mathbb{R}^d$, so that Gaussian processes can be applied.

Then, two key elements are needed to profile the convergence of an auto-tuner:

- A known good configuration.
- A log of behaviour of the objective function for that configuration.

As described in Section 2.2, there are two types of problems which may occur during a Bayesian optimisation. The first step in improving the convergence of an auto-tuner is to diagnose which of these issues is at fault for the poor convergence.

- **The model fails to capture the objective function after a reasonable number of iterations.** This can be diagnosed in two ways. In both cases, we compare the model's prediction with the objective function's behaviour. First, by comparing them on the known good configuration, using the objective function log. This allows us to diagnose why good configurations are undervalued by the model. Second, by comparing them on the configurations selected by the optimisation. Here, we can diagnose what makes the model optimistic about sub-optimal configurations. If a structured model is being used, then we can compare each model part's prediction with the corresponding observed behaviour, and hence diagnose the part of the model that is at fault.
- **The numerical optimisation fails to find promising configurations.** This can be diagnosed by evaluating the acquisition function of the known good configurations and comparing it with the acquisition function of the best configuration found by the numerical optimisation. If the known good configurations have higher acquisition function value, then the numerical optimisation failed to converge. For example, in Section 6.3, I mentioned that the DIRECT optimisation algorithm sometimes failed to find configurations with high expected improvement on the Branin-Hoo benchmark. I observed this by comparing the expected improvement of the known optimum with the best one found by DIRECT.

There are two further issues that can happen when extending a Bayesian optimisation to a structured Bayesian optimisation:

- **The `ProbEngine`s fail to converge to adequate posterior distributions.** This can happen if the number of model parameters being sampled is too large when compared with the number of particles used. Typically, this may happen if more than five parameters are sampled in the model class's constructor. An easy way to diagnose this is to train multiple `ProbEngine`'s with the same data and see if there predictions differ. If they do, it is likely the `ProbEngine`s have degenerated and their predictions are only based on few particles.

- **The numerical optimisation stage becomes more expensive than the objective function.** This can happen through decompositions if there are many nested optimisations.

In the rest of this section, I list the different techniques that a developer can employ and refer to the problem they tackle. This offers a summary of most of the methods presented in this dissertation. I first discuss techniques to improve the model's convergence. These techniques

Increase the similarity of the model to the objective function. This was the topic of Chapter 4. By capturing the developer's true prior about the shape of the objective function, we are able to converge towards the true objective function with less empirical data.

Improve the model parameter's prior distribution. A developer should check that the inferred model parameters are sensible and not too far from the prior's spectrum. They can do so by measuring the average values of the model parameters after a few iterations. This both improves the similarity of the model to the objective function and makes it easier for the `ProbEngine` to converge to the correct posterior distribution.

Make the model finer grained and increase the number of runtime measurements used. Once again, this will reduce the number of iterations needed for the model to converge towards the objective function's response surface. If there are already multiple parts in the model, the developer should first diagnose which part is failing to converge. Then, refine the models at fault.

Use cheap experiments. If the model converges too slowly, and some cheap experiments are available and instructive about the behaviour of the objective function, then they can be used in the manner described in Section 6.4. The number of iterations on the inner Bayesian optimisation should be tuned. One heuristic to do so is to make the optimisation spend at least half of its time executing the overall objective function. Hence, we can guarantee that the optimisation is at worst two times slower than without the use of cheap experiments.

If the numerical optimisation fails to converge, two main techniques can be used.

Perform the numerical optimisations on samples from the model, and discriminate between them using the expected improvement. This technique was discussed in Section 6.3. By making the numerical optimisation work on samples of the model, we are making it work on a smoother objective function. This in turns makes it more likely to converge to good values.

Decompose the numerical optimisation of the parameter space. Off-the-shelf numerical optimisation algorithms will perform poorly on configuration spaces with more than five parameters. Decompositions will help tackle the independence between parameters which will make the optimisation more likely to converge.

If `ProbEngine`s fail to converge, two techniques can be employed.

Increase the number of particles used. A typical number of particles is 10,000 but larger numbers can be used. In particular, one can use much larger number of particles for the first few observations and reduce their number later on. This way, the model's prior is sampled with a large number of particles, most of which are discarded based upon the first measurements.

Exploit the independence between different parts of the probabilistic model. This was described in Section 4.2.4. This will help `ProbEngine`s converge to the correct posterior distribution. Runtime measurements are necessary to exploit the conditional independence in the model.

Decompositions may lead to a long numerical optimisation time. There are two approaches to reduce this time.

Use Bayesian optimisation and reinforcement techniques in the numerical optimisation. A developer can reduce the number of iterations needed by decompositions by using the structured Bayesian optimisation and reinforcement learning based techniques described in Sections 5.4 and 5.5. This may reduce the cost of the numerical optimisation.

Tune the number of Thompson samples used by the numerical optimisation. There is a trade-off between the quality of the configurations evaluated on the objective function and the duration of the numerical optimisation. Performing the numerical optimisation on more samples will increase the quality of the best one found at the cost of a higher overhead per iteration. It makes sense to try and balance the time spent in each of the two stages. One extreme example of this is the sort case study in which the numerical optimisation takes orders of magnitude longer to execute than the objective function. This is due to the difficulty of optimising decision trees. Hence, in that context I use a single sample per iteration.

Together, these techniques offer ways to tackle the different issues that may occur with a bespoke auto-tuner. In this dissertation, I have presented a number of abstractions which makes the implementation of these techniques simple and intuitive. In the next chapter, I evaluate bespoke auto-tuners built using these techniques on the three case studies. This will allow me to defend the thesis of this dissertation – that using a small amount of domain specific information allows the construction of bespoke auto-tuners with higher performance than off-the-shelf optimisers.

6.6 Summary

BOAT includes a number of tools and abstractions. In this chapter, I showed how to combine them to build a bespoke auto-tuner. Implementing a structured Bayesian implementation requires declaring a number of components:

- **Configuration space**: the domain of the optimisation.
- **Structured probabilistic model**: the expected behaviour of the objective function.
- **Sub-optimisation**: a procedure to find the next configuration to evaluate. I argued in this chapter that this was best achieved by generating Thompson samples and discriminating between them with an acquisition function, such as the expected improvement.
- **Objective function**: the function being optimised.
- **Learning function**: a procedure performing inference on the probabilistic model given a new measurement.

Furthermore, I proposed a methodology to exploit cheap experiments by making them part of the objective function of a sub-optimisation. A core advantage of this approach is that the schedule of experiments is explicit, and hence easy to tune.

Finally, I discussed a methodology to build bespoke auto-tuners. The core idea is to start with an overly naive auto-tuner, and then iteratively refine its components so that they are more adapted to the problem at hand.

CHAPTER 7

Evaluation

In this chapter, I evaluate the use of BOAT via the three case studies originally presented in Section 3.1.1: *(i)* the garbage collection case study in which I tuned configuration flags to minimise the tail latency of a database, *(ii)* the sort case study in which I tuned decision trees which dynamically dispatch arrays to appropriate sort procedures and *(iii)* the neural network case study in which I tuned the scheduling of the training of a neural network on a distributed cluster.

The case studies were designed to show a diverse range of contexts in which auto-tuning is beneficial. They presented different challenges. The garbage collection case study showed a situation in which off-the-shelf auto-tuners are amenable. Hence, the goal was to demonstrate that bespoke auto-tuners improved convergence even in these simple settings. The sort case study had a configuration space with many dependencies. It therefore required the advanced use of decomposition methods. Finally, the behaviour of the system in the neural network case study had many interacting components. Thus, it involved building an elaborate probabilistic model.

For each case study, I built a bespoke auto-tuner using BOAT and evaluated its convergence in a range of settings. This evaluation focuses on quantifying two properties:

1. **The benefits of auto-tuning.** Showing that one-size-fits-all configurations yield sub-optimal performances.

2. **The need for a bespoke auto-tuner.** Showing that my auto-tuners reduce convergence time when compared to off-the-shelf optimisers. I compare my performance with OpenTuner [AKV$^+$14] which dynamically adapts its optimisation algorithm to ones that perform well, and Spearmint [SLA12] which implements traditional Bayesian optimisation.

7.1 Garbage collection case study

The garbage collection (GC) case study is the simplest of the three case studies presented in this chapter. It only has three configuration parameters. Hence it did not require the use of decomposition methods to perform the numerical optimisation. The goal of the case study was to minimise the 99th percentile latency of a database application for a range of workloads. Section 7.1.1 shows the design of the BOAT auto-tuner which is then evaluated in Section 7.1.2.

This case study was designed and executed in collaboration with Michael Schaarschmidt who narrowed the original experiment and configuration space and implemented a first version of the optimisation.

7.1.1 Design of the BOAT autotuner

This subsection describes the construction of the BOAT auto-tuner used in this case study.

Configuration space. I tuned the *young generation size*, *survivor ratio* and *max tenuring threshold* flags of the CMS collector [YL96], which is used by default by Cassandra.

Objective function. I configured a single 8 core node to run Cassandra [Apa16] with a 8 GB fixed heap space to model a medium-sized web application. I measured the 99th percentile latency using the Yahoo! Cloud Serving Benchmark (YCSB) a popular and well recognised database evaluation benchmark [CST+10] which has become the standard for the benchmarking of NoSQL databases [RGVS+12, AB13, KGE+15]. YCSB was run on a 24 core machine co-located in the same network. Each experiment was run for 15 minutes. I ran three separate optimisations using YCSB workloads A (50% reads, 50% updates), B (95% reads, 5% updates) and D (95% reads, 5% inserts) (workload C has 100% reads and is not GC-sensitive). YCSB workloads accessed 5 million records.

Model. My probabilistic model, introduced in Section 3.2.1, is composed of three semi-parametric models. It predicts the rate and average duration of minor GCs as a function of the flag values. It then predicts the 99th percentile latency as a function of these statistics as well as the flag values. My analysis showed that the frequency at which major GCs occurred was too low to have an impact on 99th percentile latency.

I included in the parametric part of each semi-parametric model some intuition about the behaviour of the system. The GC rate model, described in section 4.3, models the rate of GCs as inversely proportional to the size of the *eden* heap region. I also found that the duration of minor GCs tends to increase with the size of the *eden* and the max tenuring threshold parameter. This was declared in the GC duration model's parametric part. The 99th percentile latency tends to be affected by two properties of GCs: their average duration and the fraction of time spent in GCs. The parametric part of the latency model

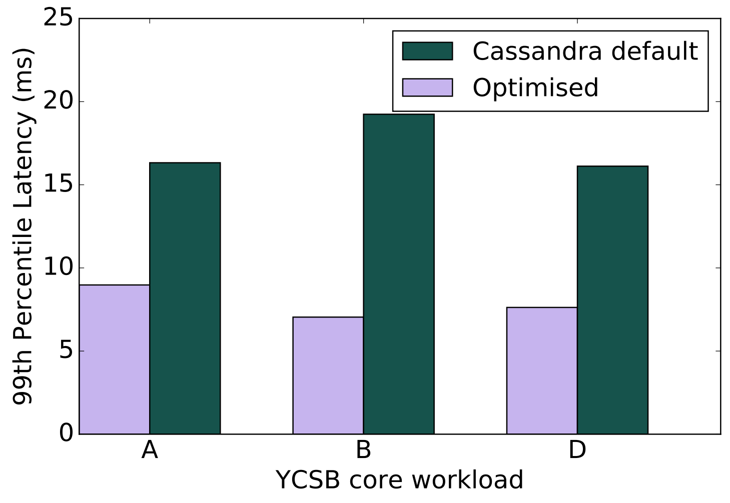


Figure 7.1: Results for YCSB workloads A, B and D.

includes linear penalties for each of these two quantities. These models are too simplistic to capture the full underlying behaviour of the computation, but they do grasp the overall trend. This is sufficient to make the BOAT-based optimiser rapidly converge towards high performance areas.

Numerical optimisation. The configuration space only has three parameters and hence I do not use decompositions. I do use the approach of Section 6.3, in which I use Thompson sampling to generate multiple good configurations and select the one which maximises the expected improvement.

7.1.2 Results

Comparison with Cassandra's default configuration. I ran the bespoke auto-tuner for 10 iterations. After each optimisation, I re-evaluated the optimised configuration and compared its 99th percentile latency with the default Cassandra configuration. The results are shown in Figure 7.1. Error bars are too small to be displayed in the figure as standard deviations were consistently below 1 ms (all results averaged over 3 runs). The optimised configuration outperforms the Cassandra default configurations by up to 63%. All optimisations converged to within 10% of the best found performance by the second iteration, after 30 minutes.

I found that the optimised configuration used large eden size, making minor collections longer but less frequent. After inspection I noted that this effectively improved the batching of the collection, therefore reducing the total work. Optimised configurations spent well under 1% of their time performing minor collections, whereas in Cassandra default configuration's case, this was around 4%.

Comparison with other auto-tuners. The previous results show tuning does yield

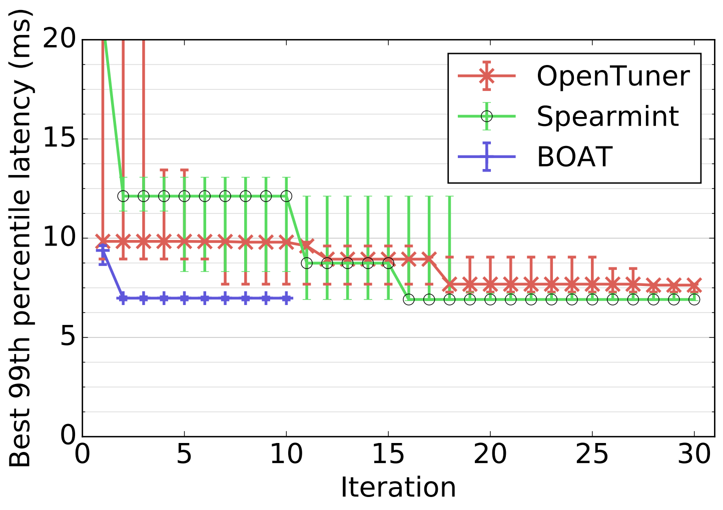


Figure 7.2: Convergence of the frameworks on workload B.

performance improvements over the default Cassandra configuration. I now consider whether generic auto-tuners would be able to yield similar performance improvements in the same timescale. Figure 7.2 compares the bespoke auto-tuner's performance with OpenTuner [AKV$^+$14] and Spearmint [SLA12], which I ran for thirty iterations. I ran each optimisation three times. For each iteration, I report the median, min and max of the best 99th percentile latency achieved so far. We see that within two iterations, the bespoke auto-tuner consistently found a high performance configuration, after thirty minutes of optimisation. In contrast, it is only at the 16th iteration that one of the other framework's median value reaches a good performance, after four hours of optimisation.

This case study shows that adding only a little structure to an optimiser can reduce its convergence time. The full probabilistic model fits in under 100 lines of C++ code. The next section presents the sort case study, which required more structure in order for the optimiser to converge.

7.2 Sort case study

In this case study, I optimised a sorting procedure based on `std::sort`. The `std::sort` implementation consists of a hybrid implementation of *insertion sort* and *quicksort*. A single parameter specifies the balance between the two algorithms. As noted by Ansel et al. [ACW$^+$09], the optimal value of this parameter is dependent on the underlying hardware architecture. Furthermore, because insertion sort performs better on almost sorted arrays, the optimal parameter value is also dependent on the input array.

To tackle a complex input distribution, where some of the arrays may be almost sorted and others not, I defined a decision tree parameter space, as presented in Section 3.3.2.

Queries of the tree can scan the first elements of the input array, at a computational cost, and inspect how sorted they are. Leaves of the tree specify a value for `block_size`. The optimal tree configuration is dependent on the input distribution of arrays and the underlying hardware.

Configuration space. I optimised decision trees of maximum depth 4. Queries at each branch of the tree have two configuration parameters. First, `scan_length` which sets the length for which an input array will be scanned by the query. When scanning an array, the query counts the number of adjacent pairs of elements that are unordered. Second, `threshold` which is compared with the number of unordered of elements counted so far – from the root of the decision tree. If an array has more unordered elements than threshold, it is dispatched to the right sub-tree. Otherwise, it is dispatched to the left sub-tree. The leaves contain a single parameter `block_size` specifying the point at which quicksort will stop recursing and insertion sort will be used instead. For simplicity, `block_size` is constrained to be a power of two.

Objective function. For each experiment, I used 1000 arrays as a representation of an input distribution. My procedure to generate random arrays takes as input three parameters: λ, x_m and α and generates each array independently. It samples the length of the array from $\mathit{length} \sim \mathit{Poisson}(\lambda)$ and samples a parameter $\mathit{rd\_changes} \sim \mathit{Pareto}(x_m, \alpha)$. It then generates a sorted array of that length and applies *rd_changes* random mutations to it.

When evaluating a configuration's performance, the objective function outputs the decision tree to code, compiles it, and runs it on the input distribution. For each array, it measures the total time of the procedure as well as the time to sort the array at the leaf.

Model. The model predicts the average time required to sort the 1000 arrays for a given configuration. It does so by modelling each array's performance individually. It models the path of arrays down the decision tree, predicting the time required to perform the scan at each branch using a linear model called `ScanModel`. At the leaves, it predicts the time to sort an array using a semi-parametric model called `SortModel`. The parametric part of `SortModel` is only concerned with the quadratic part of insertion sort. It predicts the array's sort time as a function of u_{array}, the number of unordered elements in the array, and bs_l, the value of the `block_size` parameter at the leaf:

$$\mathit{time}(bs_l, u_{\mathit{array}} \mid \theta_1, \theta_2) = \theta_1 \; bs_l^{\theta_2} \; u_{\mathit{array}}$$

where θ_1 and θ_2 are two parameters of the probabilistic model. The non parametric model uses a treed Gaussian process, as described in Section 4.5, to model the difference between the parametric model and the observed sort times. It takes as input u_{array}, bs_l and the length of the array.

Note that very little structure was added into the parametric part of `SortModel`. I initially

	length	x_m	α
Distribution I	10000	2	0.5
Distribution II	10000	500	1.5
Distribution III	30000	20	0.7
Distribution IV	100000	2	0.5
Distribution V	100000	100	0.5
Distribution VI	30000	Random shuffle	

Table 7.1: Hyper-parameters of each input distribution.

designed the model to be more complex, taking into account the computational complexities of insertion sort and quicksort. What I found was that this model performed poorly as the true execution procedure was too intricate to be accurately modelled parametrically. This illustrates the point that, when designing a bespoke auto-tuner, one should always start with the simplest possible implementation and add structure incrementally. In the end, the parametric model I settled on was sufficient to prevent the optimisation from exploring very poor regions of the configuration space. Configurations executing insertion sort on unsorted arrays were discarded. However, it did little modelling work.

Numerical optimisation. The numerical optimisation of the decision trees is the most complex one of the three case studies. It effectively performs Thompson sampling. First, it samples a model and uses it to predict – for each array and each valid `block_size` value – the associated sort time.

Then, I used the approach inspired by reinforcement learning that was described in Section 5.5 and evaluated in the context of this case study in Section 5.6. The procedure uses a probabilistic model which predicts the best average sort time achievable by a tree, given a distribution of arrays and a maximum depth for the tree. I designed a semi-parametric model for this task. It takes as input a list of arrays along with their predicted sort time for each `block_size`. It computes the best possible average sort time – if each array used its best `block_size` value – and the worst possible sort time – if all arrays were forced to use the same `block_size` value. Then, based on statistics of the array distribution and the maximum depth of the tree, it predicts a value between these two bounds.

As explained in Section 6.3, I used a single model sample per iteration of the Bayesian optimisation, effectively performing Thompson sampling. This was done to reduce the cost of the numerical optimisation which dominates the overall optimisation. The exploration properties of the procedure were therefore worse than that of the other two case studies. Some of the configurations selected for evaluation were known to be unlikely to perform well. Hence, I ran each optimisation for twenty iterations.

Comparison to `std::sort`. I applied my auto-tuner to arrays generated by the seven distributions presented in Table 7.1. Figure 7.3 shows the normalised average sort time of each optimised implementation on each distribution.

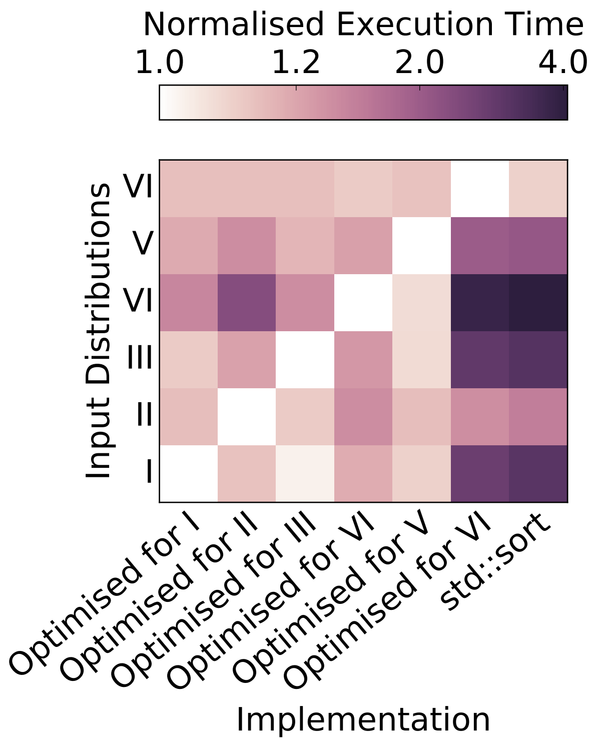


Figure 7.3: Normalised average sort time of the optimised decision trees on each array distribution. Times are normalised per row, using the best achieved average sort time for that distribution. Results obtained on an Intel Core i7-3770, with 8 hyper-threads.

It is worth noting three points. First, the optimised configuration systematically outperforms `std::sort` (right column). This is because `std::sort`'s implementation was tuned to be efficient on purely random arrays, and hence it fails to achieve optimal performance in other contexts. Even for purely random arrays (Distribution VI), its performance is sub-optimal because its `block_size` value is not adapted to the underlying machine. The decision tree optimised on Distribution VI had no branches. In its single leaf, it had the optimal value for `block_size`.

Second, for each input distribution, the corresponding optimised implementation had the best performance (the white diagonal), confirming that we were able to tune the decision trees accordingly. Third, no implementation performed well on all input distributions (no white column) suggesting a "one size fits all" approach is not appropriate here.

Comparison with other auto-tuners. Figure 7.4 compares the bespoke auto-tuner's performance with OpenTuner [AKV$^+$14] and Spearmint [SLA12], which I ran for sixty iterations on array Distribution IV. I ran each optimisation three times. For each iteration, I report the median, min and max of the best average sort time achieved so far. We see the bespoke auto-tuner quickly converged to good configurations. On the other hand, the off-the-shelf auto-tuners still had high variance and poor median performance after 60 iterations.

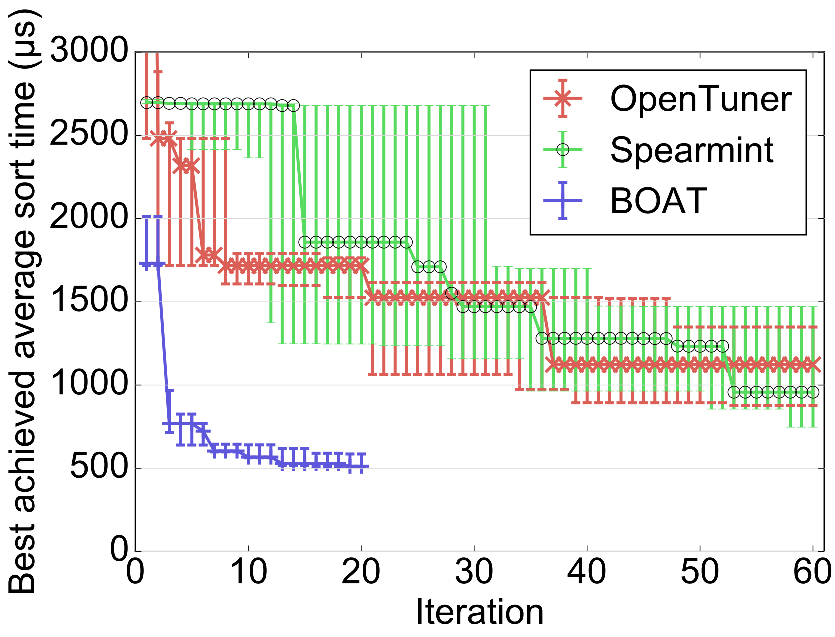


Figure 7.4: Convergence of the frameworks on array Distribution IV. Experiments were carried on an `m4.xlarge` EC2 instance.

I discuss two aspects of this experiment. First, the variance of the bespoke auto-tuner best achieved performance. There is a significant difference between the median best achieved performance (512 µs) and the maximum best achieved performance (585 µs). I attribute it to the variance due to EC2's virtualisation. The optimisation that achieved the worst result also performed significantly worse than the other two optimisations on the first iteration, despite them using the same initial configuration.

Second, the total runtime of the optimisation. The BOAT based auto-tuner suffers from an overhead of approximately 15 minutes per iteration due to the numerical optimisation. This dominates the overall optimisation cost. An entire optimisation took approximately six hours to complete. However, the other auto-tuners ran in approximately the same time. Their total optimisation time was around four hours. This is because they evaluated configurations which applied insertion sort to large unsorted arrays. Hence, I expect that my bespoke auto-tuner would finish its optimisation significantly faster than the other frameworks when applied on a distribution with larger arrays, or fewer almost sorted arrays.

This concludes the sort case study. The main take away message from this case study is that complex decomposition approaches were successful at finding efficient configurations. When initially tackling this case study, I had considered the use of heuristic tree optimisation methods. However, the results were significantly poorer. In particular, optimised implementations often did not perform best on their input distribution. Hence, the equivalent of Figure 7.3 did not show a white diagonal.

7.3 Neural networks scheduling case study

I now present my neural network case study. Neural networks have seen a surge of interest in recent years, and many frameworks have been proposed to facilitate their training [CKF11, The16, JSD+14, AAB+15, CLL+15]. In this case study, I built a bespoke auto-tuner on top of TensorFlow, a recent framework for distributed machine learning [AAB+15].

The API offered by TensorFlow to machine learning applications is low-level. Users must manually set which of their available machines should be used and how much work each should do. TensorFlow offers no automated approach to balance workloads in a distributed setup. This task is especially difficult in heterogeneous settings, where the optimal load of a machine depends on its computational power. Further, the synchronisation cost of machines can be high, and hence the slowest workers should not be used at all.

Using BOAT, I built a bespoke auto-tuner that balances a TensorFlow workload. The tuner takes as input a neural network architecture, a set of available machines and a batch size (an algorithmic parameter described in the next subsection). It then performs ten iterations, each time evaluating a distributed configuration, before returning the one with highest performance. The tuning always finishes within two hours.

The next subsection gives a background of the computation used to train neural networks in a distributed setting.

7.3.1 Training Neural Networks with Stochastic Gradient Descent

Stochastic Gradient Descent. Neural networks are typically trained with backpropagation using Stochastic Gradient Descent (SGD). Each iteration, a random batch of samples from the training set is drawn. The number of samples is called the *batch size*. Using each sample independently, an estimate of the gradient of the network parameters with regard to a *loss function* is computed. This is done via backpropagation. Gradient estimates are aggregated and used to update the neural network parameters.

Higher batch sizes lead to more parallelism, but lower batch sizes can result in better accuracy of the final neural network [Kri14, KMN+16]. We cannot quantify in advance the impact of the batch size on accuracy, as it depends on the training data, but the experiments presented in this section expose the trade-off between batch size and computational speed.

Distributed SGD. The parameter server architecture [DCM+12] typically used for distributed SGD employs two types of tasks:

- **Parameter Server** tasks synchronise the gradients at every iteration and update the neural network's parameters. Each parameter server task is associated with a section of the parameters, e.g. the first layer.
- **Worker** tasks compute the gradient estimates. Each worker is assigned a set of inputs. Each iteration, the worker fetches the updated parameter values from the parameter servers. It then computes the gradient estimate using the inputs it has been assigned. Finally, it sends these gradients back to the relevant parameter server.

Typically, stochastic gradient descent is implemented synchronously. A barrier after each iteration forces workers to compute gradient estimates using the same parameter values. Some systems implement asynchronous SGD [RRWN11], which removes this barrier and lets workers compute gradients on stale parameter versions. This improves computational performance, especially with large numbers of workers, but can hurt convergence and decrease the final result quality [CMBJ16]. In this case study, I only consider the synchronous version.

7.3.2 Tuning Distributed Stochastic Gradient Descent

When used to train neural networks, SGD can take hundreds of thousands of iterations, requiring days to complete. However, since the same computation is at performed every iteration, we can evaluate the computational performance of an SGD implementation by only running a few iterations. The goal in this case study is to optimise the distributed scheduling of a neural network in order to minimise the average iteration time, in a total optimisation time that is small compared to the training time.

Configuration Space. I tuned the scheduling of a parameter server architecture to minimise the average iteration time. Given a set of machines, a neural network architecture and a batch size, the auto-tuner optimises:

- **work**: vector of boolean parameters, specifies which subset of machines should be used as workers.
- **ps**: vector of boolean parameters, specifies which (possibly overlapping) subset of machines should be used as parameter servers.
- **inputs**: number of inputs allocated to each worker and device. Given a working machine m, $inputs_m$ refers to the number of inputs allocated to that machine. The total number of inputs across all machines must sum up to the batch size. Furthermore, given a device d on a worker machine, $inputs_d$ refers to the number of inputs allocated to that device. For a working machine m, the sum of inputs allocated to its devices must sum up to $inputs_m$.

There are effectively two boolean configuration parameters per machine specifying whether it should be a worker and/or a parameter server. There are also one to two integer parameters per machine, depending on whether it has a GPU, specifying its number of inputs. In my experiments, I tuned the scheduling over 10 machines, setting 30-32 parameters. In the largest experiment there are approximately 10^{53} possible valid configurations, most of which lead to poor performance.

Note that the auto-tuner only considers system parameters. The computation performed will be the same independently of the configuration used. In particular, the selected configuration does not affect accuracy.

Objective function. To measure the performance of a setting, the objective function first turns the configuration into python code executing TensorFlow. Then, it performs twenty iterations of SGD. The first few iterations often show a high variance in performance and hence it returns the average time of the last 10 iterations. I found this was enough to get accurate measurements, and that repeating configurations showed little underlying noise.

7.3.2.1 Probabilistic model of distributed SGD

The probabilistic model takes as input three arguments: **work**, **inputs** and **ps_size**. The first two are the configuration parameters described above. The vector **ps_size** contains floats specifying the size of the parameters hosted on each machine. This is 0 for non parameter server machines.

This is more informative than **ps**, the set of machines used as parameter server, which is what the auto-tuner directly controls. When given the list of parameter server machines, the python framework attempts to distribute the load in a uniform way. However this is still done at the granularity of individual tensors and hence some imbalance remains. For any setup, the python framework can be queried to get the resulting load distribution.

The model constitutes of three parts.

Individual device computation time. For each device – CPU or GPU – on a worker machine, I modelled the time needed for it to perform its assigned workload. This time should be near linear with respect to the number of inputs. However, SIMD architectures benefit from parallelism in the computation and hence, as the number of input grows, the performance per input of the computation sometimes increases. I explore this behaviour in more details in Section 7.3.4.2. I used a Gaussian process to model the time per input as a function of the number of inputs.

$$computation\_time(d) = inputs_d \times GP_d(inputs_d)$$

I fitted one individual device model per type of device available (e.g. c4.4xlarge CPU, or

Nvidia GPU K520).

Individual machine computation time. For machines with multiple devices, the gradient estimates were summed locally on the CPU before being sent to the parameter servers. The cost of transferring data to and from the GPUs and aggregating the gradients is non negligible and hence I modelled it as well. I used a Gaussian process to map the difference between the maximum time needed by any device on the machine and the total time needed by the machine. For a machine m

$$\mathit{computation\_time}(m) = \max_{d \in m} \mathit{computation\_time}(d) + GP_m(\{\mathit{inputs}_d : d \in m\})$$

Communication time. I modelled the communication time as a semi-parametric model. The parametric model learns a *connection_speed* parameter per type of machine (e.g. EC2 instance type). It predicts the total communication time as

$$\max_{m \,\in\, machines} \frac{\mathit{transfer}(m)}{\mathit{connection\_speed}_m}$$

where *transfer*(m) is the amount of data that must be transferred each iteration by machine m. It can be computed using:

$$\begin{aligned}\frac{1}{2}\mathit{transfer}(m) =& ps_m \times (|\mathbf{work}| - \mathbb{I}(work_m)) + \\ & \mathbb{I}(work_m) \times (\mathit{network\_size} - ps_m)\end{aligned}$$

where $|\mathbf{work}|$ is the total number of workers and *network_size* is the size of the entire neural network. The first terms model the amount of data m must transfer as a parameter server. The second, the amount of data it must transfer as a worker. The preceding $1/2$ models the fact that there are two transfer windows per iteration, one to send neural network parameters, the other to send gradients.

I used a single communication time model for the entire cluster. Finally, the model predicts the total time of an SGD iteration as the sum of the maximum predicted individual machine time and the communication time. The full model fits under 500 lines of C++ code.

Since the probabilistic model simulates individual device and machine computation times, it benefits from real measurements of these properties. The objective function therefore also measures, for each experiment, the time needed by all devices and machines to perform their assigned workload.

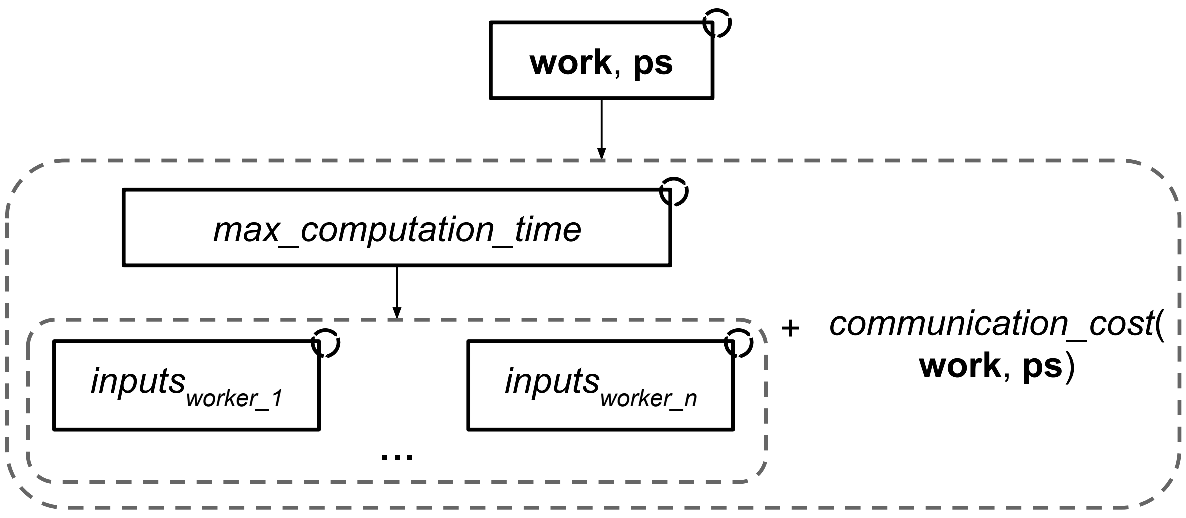


Figure 7.5: Top decompositions of the neural network case study.

7.3.2.2 Numerical optimisation

At each iteration, the numerical optimisation Thompson samples 100 configurations. It then selects the one with the best expected improvement. To perform the numerical optimisation on a model sample, I used a hierarchical decomposition.

On the highest level, I used a simulated annealing procedure to optimise the boolean values of the **work** and **ps** configuration parameters, which specify which machines are workers and which are parameter servers. This is illustrated in Figure 7.5. I made the objective function of simulated annealing use memoisation to avoid repeating the same sub-optimisation multiple times.

On a second level, I optimised the number of inputs allocated to each machine. I did so using the decomposition technique for load balancing described in Section 5.3.2, which itself uses two levels. It creates a fictitious parameter *max_computation_time* and optimises its value. The loads of the workers are then optimised individually, so that each worker processes as many inputs as possible while still performing its computation in a time shorter than *max_computation_time*'s value. *max_computation_time* can be optimised with a simple binary search. If all the inputs get allocated it is reduced, otherwise it is increased.

Finally, on the bottom level of the decomposition, I optimised the load balance between different devices on a worker using the DIRECT algorithm. Note that the load balance optimisation technique that I use across workers would not work here as the device aggregation model may predict that, for example, it is beneficial for the GPU to finish its computation before the CPU.

Instance Type	# Hyperthread	GPU	# per setting		
			A	B	C
g2.2xlarge	8	1 K520	0	1	2
c4.2xlarge	8	/	6	3	2
c4.4xlarge	16	/	2	3	2
c4.8xlarge	36	/	2	3	4
Total			10	10	10

Table 7.2: Machine and setting specifications.

7.3.3 Results

Experimental Setup. I evaluated the auto-tuner on Amazon EC2 using TensorFlow version *v0.8*. There are three inputs to the tuning procedure. The machines available, the neural network being trained and the batch size.

I constructed three machine settings, described in Table 7.2, which are designed to recreate heterogeneous environments. Each contains 10 machines of varying computational power. Settings B and C contain one and two GPU instances respectively. While neural networks perform most efficiently on GPUs, I tried to design realistic settings where a variety of CPUs and GPUs were available.

I evaluated each of the three hardware setting with the three neural networks referenced in Table 7.3, using four batch sizes for a total of thirty-six experiments. The four batch sizes for each network were selected to explore the trade-off with processing speed. Recall that batch size is an algorithmic parameter equalling inputs per iteration, and that lower batch sizes tend to improve final result accuracy at the cost of less parallelism.

A few trade-offs are important to note:

- The size of the neural network varies by almost an order of magnitude from GoogleNet(26.7MB) to AlexNet(233MB). Since the parameters of the network need to be exchanged between parameter servers and worker machines every iteration, we should expect this to have an influence on the communication part of the protocol. Specifically, we should expect that configurations optimised for the AlexNet network use fewer workers than those optimised for GoogleNet.
- The type of the network has an impact on its performance. While the GPUs instances used systematically outperform the CPU instances, perceptron architectures tend to show a smaller gap in performance than convolutional ones. Hence we should expect configurations tuned for SpeechNet to make more use of CPU machines than those optimised for the similar sized AlexNet.

I reference experiments using a shortened name, for example, C-AlexNet-2^8 refers to the experiments of scheduling AlexNet on Setting C with a batch size of 2^8.

Neural Network name	Input Type	Network Type	Size (MB)
GoogleNet [SLJ+15]	Image	Convolutional	26.7
AlexNet [Kri14]	Image	Convolutional	233
SpeechNet [SFD+14]	Audio	Perceptron	173

Ops (Millions)	Batch size range
1582	$2^6 - 2^9$
714	$2^8 - 2^{11}$
45.3	$2^{13} - 2^{16}$

Table 7.3: The three neural networks used in the experiments. Size is the size of the parameters which must be transmitted to and from workers each iteration. Ops is the number of floating point multiplications per input. The name "SpeechNet" is introduced here for clarity – this network was recently proposed for benchmarking [Chi].

Comparison with simple configurations. To show the importance of tuning, I compared the optimised configurations with two simple configurations

- **Uniform Devices:** Time per iteration when splitting the load uniformly across devices. In this context, each machine is used both as a worker and as a parameter server.
- **Uniform GPUs:** Time per iteration when splitting the load uniformly across GPUs. This only applies to Settings B and C as Setting A does not have a GPU. In this context, only machines with GPUs are used as parameter servers. I empirically found this to be the best parameter server placement.

Figure 7.6 shows the outcome of each experiment. The optimised configurations significantly outperformed simple configurations on most experiments. The greatest improvement, in the context of C-SpeechNet-2^{16}, shows a speed-up of $2.9\times$. Looking at the results closely shows that two factors lead to better performance:

- **Better load balancing:** Comparing "Uniform Devices" configurations with the optimised configurations on GoogleNet shows this well. Both used all available devices but the optimised configurations are significantly faster due to better balancing of the workload across workers.
- **Better selection of workers:** Common wisdom says that GPUs are far superior for computation on neural networks, but in multiple cases performance was increased by over $2\times$ by adding CPU machines and tuning the system correctly.

Furthermore, there are a number of insights that can be gathered from Figure 7.6 on how the different contexts impacted the optimised configurations.

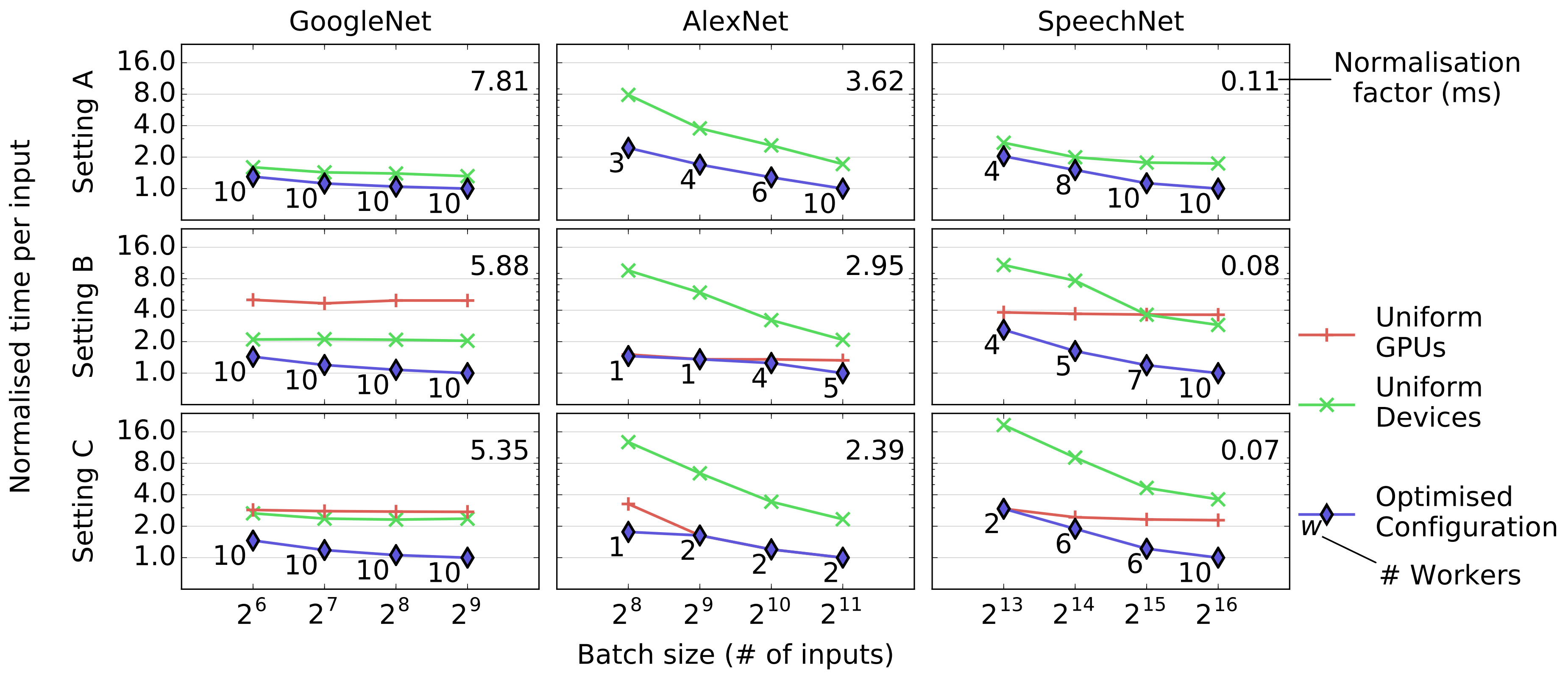


Figure 7.6: Normalised time per input (lower is better) of simple and optimised configurations on each experiment. Within each sub-graph, results are normalised by the best achieved time per input. This is always the one of the optimised configuration on the largest batch size (the lower right point of each sub-graph). The normalisation factor, i.e. the best time per input, is shown at the top right of each sub-graph in milliseconds. For each optimised configuration, I report the number of workers used.

- **Impact of batch size:** In each experiment, increasing the batch size systematically resulted in the use of a greater (or equal) number of workers. This is what is expected – increasing the batch size increases the amount of computation with no change on the communication cost.
- **Impact of network size:** We see that the optimised configuration of smaller networks, like GoogleNet, took advantage of all machines as they incurred a low communication overhead. In comparison, AlexNet and SpeechNet were only able to leverage extra machines when there was enough work to perform, i.e. when the batch size was large. GoogleNet is by far the smallest network and hence it had a low communication cost, always taking under 20% of the iteration time. In the context of the larger AlexNet and SpeechNet, this rose up to 40%.
- **Impact of the neural network type:** The optimised configurations of SpeechNet used more machines than those of AlexNet for a similar computation cost. For example, C-AlexNet-2^{11} and C-SpeechNet-2^{15} had similar performances in the GPU only setting, but SpeechNet improved significantly with more CPU machines. Part of this can be explained by the fact that SpeechNet is slightly smaller in size, and hence has a lower communication cost. The major factor, however, is the reduced gap in between CPU and GPU performance on the perceptron type of neural network architecture. The optimised configurations for AlexNet on setting B placed approximately 3.4 times more inputs on the available GPU than on each c4.8xlarge CPU. For SpeechNet on the same setting, this ratio was 1.6

Each of these items confirms the intuition that "one size fits all" approaches are not appropriate, as optimised configurations are influenced both by hardware and workload.

Finally, it is interesting to note the exposed trade-off between batch size and processing speed. Recall that batch size is an algorithmic parameter and lower batch sizes tend to produce better accuracy of the final neural network. Each sub-graph of Figure 7.6 shows how much was gained, in terms of processing rate, by increasing the batch size and hence the parallelism of the computation. With my auto-tuner, users can find optimised configurations for different batch sizes and easily explore this Pareto-front, hiding the details of the configuration used.

Comparison with other auto-tuners. I now consider whether the benefits of auto-tuning could have been achieved with an off-the-shelf optimisation tool. Figure 7.7 compares the performance of the bespoke auto-tuner with OpenTuner [AKV$^+$14] and Spearmint [SLA12], which were each ran for thirty iterations. Each optimisation was ran three times, I report for every iteration the median, min and max performance of the best configuration found so far.

The bespoke auto-tuner significantly outperformed generic auto-tuners. The median best configuration achieved by OpenTuner is 8.71s per SGD iteration, more than twice as slow

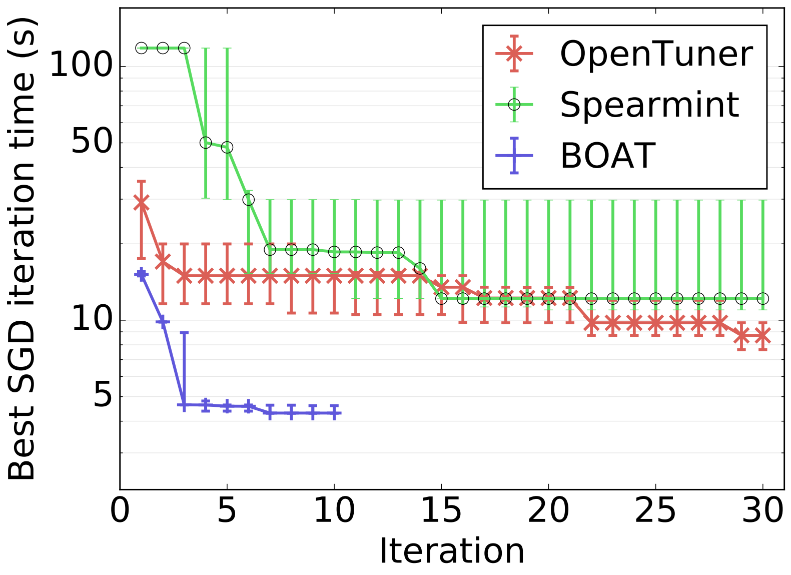


Figure 7.7: Convergence of the frameworks on C-SpeechNet-2^{16}.

as BOAT's median time (4.31s), and not much faster than the Uniform GPUs configuration (9.82s). The reason this tuning task is difficult is because the space of efficient configurations is extremely narrow – assigning one of the workers too much work creates a bottleneck and yields poor performance.

All of the experiments finished the ten iterations within two hours. As neural networks training typically lasts over a week, the performance gains largely outweigh the tuning overhead, making this auto-tuner practical in realistic settings. The largest experiments involved 32 dimensions. I expect that the auto-tuner retain the same convergence in larger settings as there would be a proportional increase in the number of runtime measurements.

7.3.4 Querying the learnt models

One of the advantages of optimising configurations with a structured probabilistic model is that, after the optimisation, we can query the model to understand the system's behaviour. In this subsection, I explore two properties of the learned models. First, I analyse the communication model inferred after the optimisation of A-AlexNet-2^{10}. Then, I query the individual machine models inferred from optimising B-GoogleNet-2^7.

7.3.4.1 Querying the communication model

A surprisingly important aspect of the tuning is the ability of the optimiser to detect machines with fast network connections and to use them as parameter servers. I investigate the impact of parameter server placement in the context of A-AlexNet-2^{10}. Table 7.4 shows the details of the optimised configuration. Note that only three of the six available worker machines are used as parameter servers. Figure 7.8 shows the breakdown of the

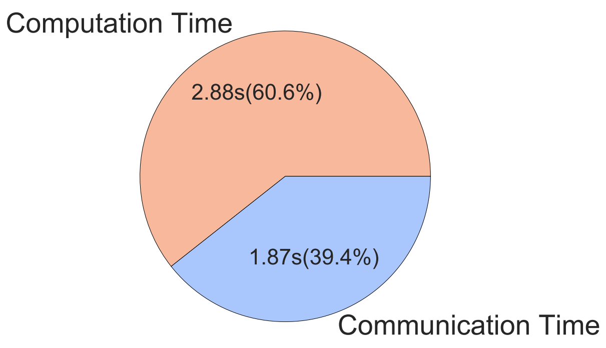


Figure 7.8: Distribution of the computation and communication costs in the optimal found configuration of A-AlexNet-2^{10}.

	`c4.2xlarge`						`c4.4xlarge`		`c4.8xlarge`	
PS	N	N	N	N	N	N	N	Y(19MB)	Y(147MB)	Y(67MB)
Work	N	N	N	N	Y(96)	Y(96)	Y(180)	Y(180)	Y(236)	Y(236)

Table 7.4: Optimal configuration of A-AlexNet-2^{10}. First row describes whether a machine is used as a parameter server – in brackets is the size of the parameters hosted on that machine. As mentioned in section 7.3.2, this is a quantity the auto-tuner can observe but does not control directly. It is dependent on the number of parameter server machines. The second row shows whether a machine is used for work – in brackets is the number of inputs scheduled on that machine.

costs for that model. Communication is an important part of the execution, taking 39.4% of the total runtime.

I am able to query the model on different configurations and observe their predicted impact. For example, I find that using all six working machines as parameter servers is predicted to increase the communication time from 1.87 to 2.74 seconds.

I investigate the parameters inferred by the model. Table 7.5 shows the distribution of network connection speeds inferred by the model against the ones advertised by Amazon for the same instances [Ama]. Although the model's inferred parameters do not directly match the advertised values, they do follow the correct trend. Furthermore, they were sufficiently accurate to make the optimisation converge to a good configuration in which the majority of the neural network's parameters were placed on instances with fast network connections.

Instance Type	Modelled time per GB(s)	Advertised time per GB(s)[Ama]
`c4.2xlarge`	4.12 ± 0.69	8.0
`c4.4xlarge`	3.25 ± 0.69	4.0
`c4.8xlarge`	2.22 ± 0.45	2.0

Table 7.5: Connection factors inferred by the model after 10 iterations of A-AlexNet-2^{10}.

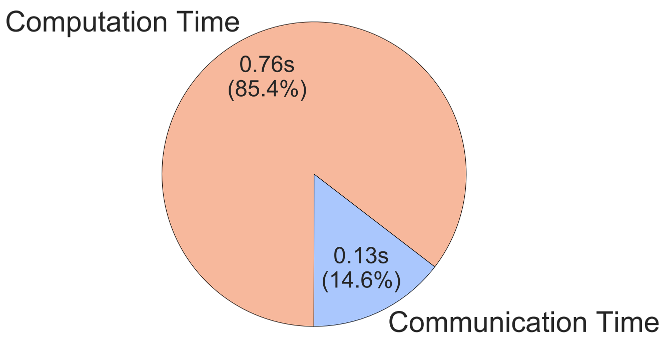


Figure 7.9: Distribution of the computation and communication costs in the optimal found configuration of B-GoogleNet-2^7.

	`c4.2xlarge`			`c4.4xlarge`		
PS	Y(5.20MB)	Y(3.52MB)	Y(3.71MB)	N	N	N
Work	Y(8)	Y(8)	Y(8)	Y(13)	Y(13)	Y(13)

`c4.8xlarge`			`g2.2xlarge`
Y(7.05MB)	Y(3.68MB)	Y(3.54M)	N
Y(12)	Y(12)	Y(12)	Y(4-CPU | 25-GPU)

Table 7.6: Optimal configuration of B-GoogleNet-128. First row describes whether a machine is used as a parameter server, and if so, the size of the parameters it holds. Second row shows whether it is used for work, and if so, the number of inputs scheduled on each of its devices.

7.3.4.2 Querying the individual machine model

Table 7.6 shows the optimised configuration for B-GoogleNet-2^7. The breakdown of costs for that setting is shown in Figure 7.9. There are two surprising features of this optimised configuration which I discuss here.

First, the parameter server placement in the optimised configuration seems sub-optimal – `c4.4xlarge` machines could be used instead of `c4.2xlarge` as they have higher bandwidth. However, it is unlikely that this improvement would have a significant impact. The best average communication time that I observed across GoogleNet experiments was 0.12 seconds, only 0.01 seconds faster than this configuration.

Second, the auto-tuner assigned 12 inputs to each of the `c4.4xlarge` machines, which is one more than the loads assigned to the more powerful `c4.8xlarge` machines. Querying the model confirmed that for low input sizes `c4.4xlarge` was predicted to have the higher performance. I investigated this by performing a series of experiments on a `c4.4xlarge` and `c4.8xlarge` instances. These results along with the model predictions are shown in figure 7.11.

The experiments confirm that the auto-tuner was being sensible when placing one more item on `c4.4xlarge` instances than on `c4.8xlarge`. For low batch sizes, the large number of cores of `c4.8xlarge` instances (36) leads to a significant overhead. This is despite there

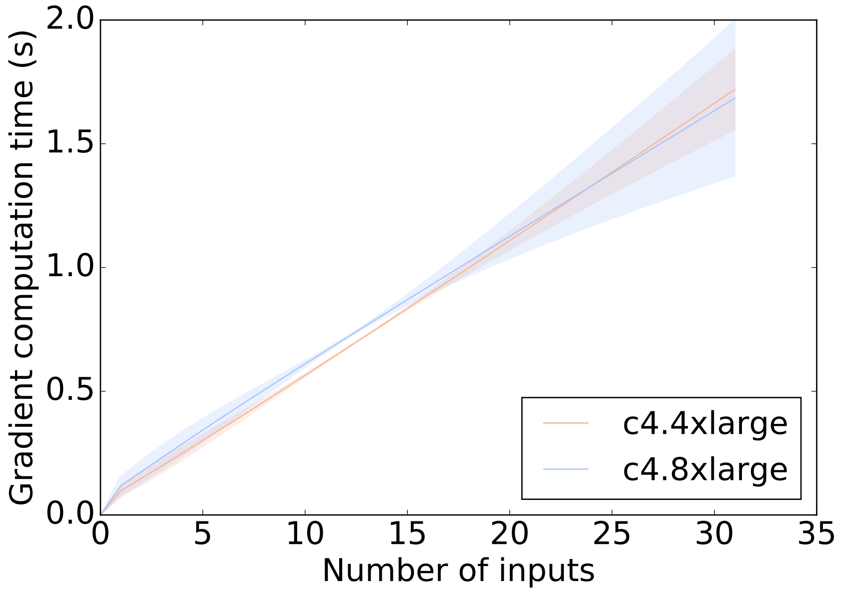


Figure 7.10: Inferred time to compute the gradient of GoogleNet by individual machines. The shaded area shows the confidence interval. Note that the model is uncertain about the regions far from the optimum. In the region of interest, 10 – 15 inputs, it gives a slight edge to `c4.4xlarge` machines.

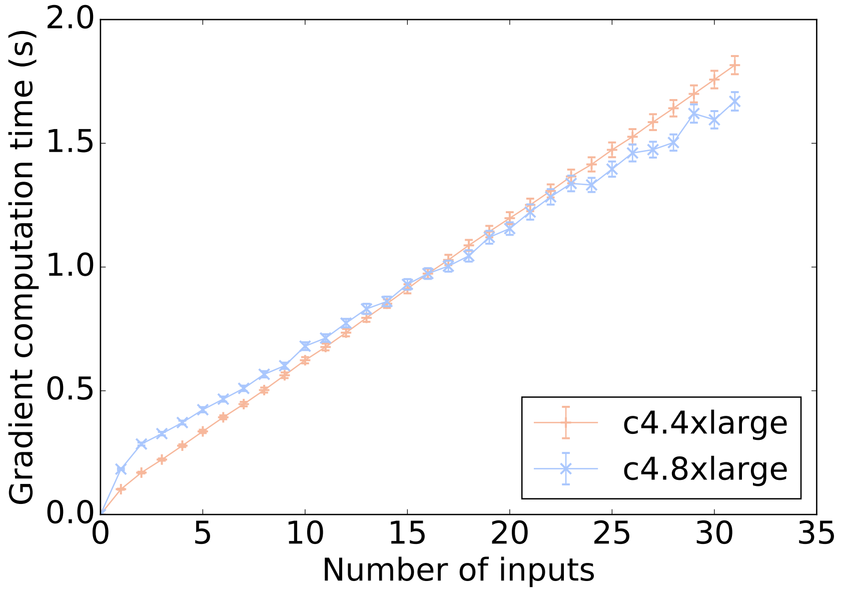


Figure 7.11: Empirical time to compute the gradient of GoogleNet by individual machines. Each data point is averaged over 50 iterations with a 10 iterations warm up. Error bars show the uncertainty of the average.

being large amounts of parallelism within the computation of the individual inputs. This highlights the importance of auto-tuning – a rule of thumb approach would have likely made a sub-optimal use of the machines available.

These results also highlight the benefits of BOAT's grey box approach. Querying the model offers an easy way to understand the decisions made by the optimiser, something that would be difficult to achieve with a standard black box auto-tuner. This can help systems developers diagnose inefficiencies and bottlenecks. In this particular example, they may reconsider their implementation of convolutional neural networks on many core machines.

7.3.5 Conclusion

In conclusion, I built a bespoke auto-tuner to optimise the distributed scheduling of the training of neural networks using BOAT. In the majority of cases, the auto-tuner improved performance over naive configurations. The auto-tuner was able to adapt to different loads, network types and hardware. When applied to this task, standard auto-tuners were not capable of converging to good configurations in a reasonable time.

An interesting outcome of the results was that, in mixed CPU and GPU clusters, using CPUs could significantly improve performance. Finally, I showed the benefits of having a structured probabilistic model of the system being optimised. I used it to understand the trade-offs tackled by the tuned configurations and diagnose inconsistencies in the system.

7.4 Summary

This chapter presented the application of BOAT to the three case studies presented in this dissertation. The main conclusions are as follows:

- In all cases, the bespoke auto-tuner surpassed the performance of its off-the-shelf counterparts. It converged in fewer iterations to high performance configurations. In some cases, the optimisation problems were too difficult for off-the-shelf auto-tuners to converge in a reasonable time. On the other hand, the bespoke auto-tuner consistently converged within ten to twenty iterations. This supports the thesis made in this dissertation that the construction of bespoke auto-tuners makes auto-tuning applicable in previously unexplored contexts.

- Optimised configurations can show significant performance improvements over 'one-size-fits-all" approaches. In the garbage collection case study, optimising garbage collection flags reduced Cassandra's 99th percentile latency by up to 69% over default settings. In the neural network case study, the optimised configurations resulted in speed-ups of up to 2.9× compared to simple alternatives.

- For simple optimisation problem, such as the garbage collection case study, adding only a little structure was shown to reduce convergence time. Hence bespoke auto-tuners can be a sensible choice in simple settings.
- For optimisation problems with complex configuration spaces, such as the sort case study, the decomposition approaches proposed in Chapter 5 were shown to be successful at tackling the numerical optimisation stage of a Bayesian optimisation.
- The probabilistic models used by BOAT can be queried to profile the system's behaviour. In the neural network case study, I used them to deduce the effective network bandwidth of different EC2 instances. I also diagnosed that, in some contexts, machines with more CPU cores had worse performance.

Together, these results show that BOAT can be used to build bespoke auto-tuners with good convergence properties. In the next chapter, I discuss similar approaches that are used in the context of computer systems.

CHAPTER 8

Related work

This chapter discusses approaches used in the context of computer systems that are related to BOAT. What distinguishes BOAT from most of the works presented in this chapter is that BOAT is designed for *iterative optimisation*. Given an objective function, a bespoke auto-tuner will repeatedly evaluate the objective function to find a good configuration. I discuss applications of iterative optimisation to computer systems in Section 8.1.

Many systems instead perform *preemptive optimisation* (Section 8.2). They receive a query at runtime and optimise some aspect of its schedule to maximise performance. This is more difficult as the runtime behaviour of a query cannot be known in advance. Hence, they rely on simple performance models, sometimes coupled with historical traces. Some systems will also continuously monitor their current performance to adapt their configuration runtime. I briefly discuss these approaches in Section 8.3.

Preemptive and runtime optimisation approaches are complementary to the ones proposed in this dissertation. Runtime decisions are necessary to respond to a changing environment. However, to solve trade-offs such as the balance between decision making and actual computation, a static approach is more appropriate. This is what BOAT aims to offer.

Section 8.4 discusses the applications of decompositions to network scheduling and optimisation. Finally, Section 8.5 briefly reviews related work in probabilistic programming.

8.1 Iterative optimisation of programs

This section discusses approaches which, like BOAT, perform an iterative optimisation. I first discuss generic auto-tuning frameworks (Section 8.1.1). I then review some uses of auto-tuning in computer systems (Section 8.1.2). Finally, I mention compilers that are able to perform an iterative optimisation (Section 8.1.3).

8.1.1 Generic auto-tuners

This subsection highlights PetaBricks [ACW+09] and OpenTuner [AKV+14], two significant contributions in the space of generic auto-tuners.

PetaBricks is a language and compiler which allows users to express algorithmic choice. Users implement a PetaBricks program which allows for a range of valid implementations. The PetaBricks compiler then performs an iterative optimisation to find a high performance implementation of the program on the target machine. The compiler turns the original program into parallel C++ code and evaluates a variety of implementations before selecting the one with the highest performance. This optimisation is performed using an evolutionary algorithm.

One of PetaBrick's features as a general purpose autotuner is that it natively reasons about algorithms that are applied on collections of elements such as arrays and matrices. It leverages this by evaluating the performance of some configurations on reduced sized inputs at a fraction of the cost.

An extension to the PetaBricks language allows it to leverage input features. This way, a decision tree can be built to dispatch inputs to appropriate configurations [DAV+15]. This is similar to the configuration space I used in my sort case study and, indeed, the authors apply the extension to the problem of sorting. However, due to the use of PetaBricks' parallel runtime, the results are not directly comparable to the ones presented in this dissertation.

OpenTuner is a python library for auto-tuning. Users implement a python class with a number of parameters that should be optimised, along with an objective function. Unlike PetaBrick, the configuration space does not allow for the recursive parameters such as trees.

To select the next configuration to evaluate, OpenTuner uses one of its many optimisation algorithms, such as evolutionary algorithms, or the Nelder-Mead method [NM65]. A top-level meta-optimiser iteratively learns which algorithms perform well on the problem at hand and select them more frequently. All configuration-evaluation pairs are stored in a database that is shared among algorithms.

One of OpenTuner's main conclusions is that the optimisation should be adapted to the problem at hand. For example, OpenTuner distinguishes between different types of parameters. Categorical parameters are represented differently from integers. When optimisation algorithms are executed, these properties are exposed. This is different from traditional optimisation which usually maps a configuration into a vector of real numbers. Similarly, the optimisation algorithm that is most used is the one that performs best on the specific problem. In both these cases, the optimisation is adapted to the problem at hand. This conclusion motivates the work presented in this dissertation. In order to perform the optimisations efficiently, we must leverage the structure specific to the problem at hand.

It is worth noting that PetaBricks and OpenTuner were both designed to optimise programs with execution times not exceeding the seconds scale. Hence, they were able to try thousands of different configurations in a reasonable time. This is different to the problems tackled in this dissertation for which the optimisation budget did not exceed twenty evaluations of the objective function.

8.1.2 Auto-tuners for system configurations

A number of auto-tuning frameworks have been developed to optimise the performance of specific systems. In these works, estimating the performance of a configuration can be done in a short time, allowing the auto-tuner to evaluate many configurations.

This approach is extensively used for numerical libraries so that they can adapt to the underlying hardware. ATLAS [WD98] generates an optimised implementation of the BLAS API [BDD+01] for linear algebra. Similarly, PHiPAC [BACD97] auto-tunes the implementation of dense matrix multiplications, SPARSITY [IY01] and OSKI [VDY05] optimise the implementation of sparse matrix kernels, FFTW [FJ98] and UHFFT [MJ01] generate optimised implementations of fast Fourier transforms and SPIRAL [PMS+04] auto-tunes a range of digital signal processing procedures. In these works, the configuration space is structured enough so that an almost exhaustive approach combined with heuristics generates good implementations in a reasonable time.

Another area which has seen large amounts of auto-tuning work is the optimisation of GPU code [SFLD15, NM09, MGJ13, GGXS+12]. For example, Steuwer et al. [SFLD15] introduce a high-level language for GPU programming. They develop a set of rewrite rules to transform programs written in their language to OpenCL code. The order in which the rules are applied changes the resulting implementation. They search the space of possible implementations by applying the rules in random, heuristically-guided orders.

It is rare to see iterative optimisation used for more complex systems as they are usually more expensive to evaluate. Chopper [HNADAD15] explores the configuration space of the Linux file system to reduce its tail latency. However, the goal was to generate an overall improved implementation, not to tackle performance portability. Duan et al. propose iTuned [DTB09], which uses traditional Bayesian optimisation to tune the configuration parameters of a database.

8.1.3 Iterative compilation

Optimising compilers use an implicit model of the runtime behaviour of the program to select an optimised implementation. Some compilers can exploit runtime measurements from previously compiled versions of the program to refine this model [PH90, BKK+98, ABC+06, CLM16]. This is sometimes referred to as *profile-guided optimisation* (PGO) or *feedback-directed optimisation* (FDO). On a first iteration, the program is compiled and its behaviour profiled. The program is then compiled again with the profile information given to the compiler. This process can be repeated for multiple iterations. The type of optimisation problems that I tackle with BOAT are more complex and profiling information is not sufficient to deduce good configurations.

8.2 Preemptive optimisation of programs

In many contexts, a system takes as input a program or query to generate a corresponding optimised implementation. This section discusses some of these approaches and their relationship to BOAT. One fundamental difference is that BOAT based auto-tuners perform an iterative optimisation and hence repeatedly measure the performance of the same program. On the other hand the approaches presented here have an implicit or explicit model of the computation behaviour, which they use ahead of time to optimise it.

Optimising compilers. Perhaps the most common use of computer program optimisation is via optimising compilers. They reason at a low-level about a program's execution to optimise its implementation [CT11]. While some of these optimisations yield strictly faster programs, such as common sub-expression elimination [Coc70], other take into account an implicit model of a machine's architecture [KA01].

Database compilers. In a similar way to traditional compilers, databases use performance models to predict the behaviour of queries before their execution [Cha98, ZLC10]. They perform an optimisation over a complex configuration space to schedule the query, but rely on the constrained scope of queries – which only use relational operators – to make accurate predictions.

Some approaches also exploit historical data. For example, Wrangler [YAK14] uses support vector machines [Mur12] trained on previous executions. It predicts ahead of time whether executing a task on a machine will lead to a straggler. Starfish [HLL+11] and MRTuner [SZL+14] profile the execution of MapReduce tasks [DG08] to optimise the configuration parameters used for subsequent tasks. Once again, the simplicity of the MapReduce framework makes it possible to accurately predict the behaviour of a query ahead of time.

Cluster scheduling. The role of schedulers is to allocate resources to tasks. They tackle the optimisation problem of allocating resources in the most performant or most fair way. In some cases, this optimisation is implicit. For example, the Hadoop fair scheduler [Had] uses a queue based approach to dispatch tasks.

Another approach is to use a cost model to evaluate the impact of scheduling decisions.

Amir et al. propose E-PVM [AAB$^+$00] which uses such a model to schedule a single job. Google's Borg [VPK$^+$15] uses a similar approach, with a different cost model designed to reduce the fragmentation of tasks across machines. The Quincy scheduler [IPC$^+$09] also uses a cost model but considers all tasks and machines simultaneously. All resources and tasks are mapped on a flow network. It uses a min-cost flow solver to find an optimal schedule. This approach more accurately captures the optimisation problem being tackled, although it is also more expensive. However, Firmament [GSG$^+$16] shows that it is possible to scale this method to modern datacenters.

The use of a cost model is similar to the approach presented in this dissertation. Both use this model to perform an optimisation and make informed choices. However, in order to make these choices in a reduced time, the models discussed here are simplified so that they can be optimised in a short time. For example, min-cost flow solvers rely exclusively on linear relationships. BOAT auto-tuners can perform optimisations over more complex and expensive models, but will not converge in the sub-second time scale.

Cheap experiment based approaches. A recent trend in systems has been to perform a reduced execution of a program to infer its long term behaviour. For example, the incoming query can be profiled on a reduced dataset or for a limited time. Then, a statistical approach can be used to predict the query's entire behaviour.

Paragon [DK13] is a data centre scheduler which uses collaborative filtering techniques [Mur12] to classify incoming workloads and schedule them appropriately. The collaborative filter models platform heterogeneity and the interference between co-located workloads. An incoming workload is executed for a few short-runs to determine its runtime behaviour. The results are then compared with previously scheduled workloads to predict how well the application will run on the different hardware platforms. In their subsequent work Quasar [DK14], the authors generalise their approach to resource allocation, using previous workload traces to predict how well an application will scale-up or scale-out.

In ProteusTM [DDK$^+$16], Didona et al. use a similar approach to select an appropriate transactional memory (TM) implementation for an application. It starts by profiling a number of applications off-line to build a training set. At runtime, when a workload is received, it uses Bayesian optimisation to explore the impact of TM implementations on the application's performance. It then uses the data observed throughout the optimisation to recommend a TM implementation, using a collaborative filter and the off-line data.

Ernest [VYF$^+$16] is a performance prediction framework for large scale data-analytics applications. They design a parametric model to model an applications' performance as a function of the quantity of data and the number of machines used. Given a data analytics application, they predict the application's performance by first running it on a subset of the data. They use optimal experimental design methodology [Puk93], including the experiment's cost, to select the cheap experiment. The parametric model then predicts the cost of running the full application based on the empirical measurements.

These later approaches use generic probabilistic models to predict performance. This works well when the domain of possible configurations is small enough. However, in high dimensional settings, the curse of dimensionality will prevent a model from accurately representing all possible configurations. BOAT tackles this by using bespoke models which are engineered to resemble the system's behaviour.

8.3 Runtime control and optimisation

In this section, I discuss systems that dynamically adapt their configuration or decisions based on runtime measurement. In order to react well to their environments, these approaches require a high quality feedback function quantifying the system's performance in real time.

JIT compilers. Some Just-In-Time (JIT) compilers are capable of dynamically recompiling the application's code based on its observed behaviour [GPF06, GES+09].

Runtime tuning frameworks and applications. Active harmony [cCH02] is an automated runtime tuning system. It allows an application to expose its configuration knobs so they can be tuned at runtime. Underneath, it uses the Nelder-Mead simplex method to search the domain of configurations.

Similarly, Heartbeat [HES+10] is a framework for runtime tuning. It decouples applications being tuned from the dynamic optimiser. PowerDial [HSC+11] uses Heartbeat to dynamically tune configuration parameters to trade-off quality-of-service metrics with power usage. Similarly, the Angstrom processor [HHK+12] is a multicore which monitors the hardware state at a low granularity. It uses the Heartbeat framework to coordinate some of its hardware properties, such as each core's voltage.

Runtime control in systems. Computer systems often monitor the performance of their assigned tasks to schedule them accordingly. ARIA [VCC11] and Jockey [FBK+12] dynamically control the resources allocated to a job so it can meet its service level objective. They do so in the context of MapReduce [DG08] and SCOPE [CJL+08] databases respectively. They use an analytical model to predict a job's remaining execution time as a function of the resources allocated to it. The parameters of the model are estimated from historical traces. This is a concept similar to the models used by database query optimisers mentioned in the previous section.

Graph processing systems perform analytics tasks, such as PageRank [PBMW99], on large graphs [MAB+10, GLG+12, RMZ13]. Typically, the computation is scheduled in a fixed way that cannot be adapted to the workload. However, Chaos [RBMZ15] uses a model of the computation at runtime to adapt its behaviour. It uses a work stealing scheme to balance the load onto multiple machines. The decision whether or not to steal is made by examining whether the model predicts this to be beneficial, based on a runtime estimate of the amount of remaining work.

Self-adapting systems. Self-adapting or self-aware systems dynamically adapt their configuration parameters, often via the use of control theory or reinforcement learning approaches [ST09]. These methods have been applied to databases [CN07], memory controllers [IMMC08], hardware instruction sets [DR14] and data structures [EWA11] among many others.

8.4 Decompositions and network scheduling.

When viewed under the lens of an optimisation problem, network protocol stacks can be shown to perform an implicit optimisation decomposition [CLCD07, KMT98]. The different layers of the stack vertically decompose a resource allocation problem, with each layer's protocol optimising a subset of the parameters. Furthermore, the protocols themselves can be shown to perform a horizontal decomposition, where the different parts of the allocation problem are optimised in a distributed fashion. In particular, the TCP/IP and MAC protocols can be interpreted in this way, and the optimisation problem they solve can be reverse engineered [WJLH06, LCC06].

The decomposition methods used in this dissertation are somewhat different. They are declared directly and hence do not rely on a protocol to implicitly optimise them.

8.5 Probabilistic programming

Probabilistic programming has seen a recent surge of interest after the development of Church [GMR$^+$08], the first Turing-complete probabilistic programming language. Since then, multiple frameworks capable of performing inference on arbitrary probabilistic programs have been proposed [MSP14, TvdMW15, TKD$^+$16]. They typically offer interfaces to a variety of inference algorithms. They mark a change from frameworks which perform inference on restricted classes of models [STB$^+$96, Plu03, CGH$^+$16, MWG$^+$]. In particular, Stan [CGH$^+$16] uses Hamiltonian Monte Carlo techniques [Nea11] to generate samples from the posterior distribution. This leads to high inference performance on some models, but restricts the domain of the inference to be a fixed set of continuous parameters.

Probabilistic-C++ is Turing-complete as it allows the use of arbitrary C++ code. It is a lightweight implementation of the particle filtering method for probabilistic programs that was recently proposed by Wood et al. [WvdMM14], and subsequently implemented in Probabilistic-C [PW14] as well as other frameworks [MSP14, TvdMW15]. Markov chain Monte Carlo is an alternative inference algorithm and was used in Church [GMR$^+$08]. Slice sampling has also been applied to probabilistic programs [RG15].

CHAPTER 9

Conclusion

The growth of complex computer systems has increased the need for high quality auto-tuners that are capable of adjusting a system's configuration parameters to a users setting, and hence provide performance portability.

As I observed in **Chapter 2**, Bayesian optimisation – one of the more advanced auto-tuning techniques – can fail in large dimensional spaces. To address this challenge, this dissertation has proposed a general framework which allows a system developer to design a bespoke auto-tuner for their system by exposing a small amount of domain specific structure.

- In **Chapter 3**, I introduced *structured Bayesian optimisation*, an extension of the Bayesian optimisation algorithm which leverages a user-given probabilistic model. Bespoke models, which are key to all techniques presented in this dissertation, allow the Bayesian optimisation to grasp the behaviour of the objective function after only a few iterations, and hence find high performance configurations. I showed how structured Bayesian optimisation was integrated within the BOAT framework. I presented BOAT's *configuration space* abstraction which enables the construction of configuration spaces with complex dependencies between parameters. In particular, it allows the construction of recursive configurations such as trees.

- **Chapter 4** was concerned with providing tools for users to model their system's behaviour. I presented *Probabilistic-C++*, a high performance probabilistic programming library. Probabilistic-C++ offers ways for users to exploit the independence and conditional independence in their model to help the underlying inference converge. I showed how Probabilistic-C++ can be used in practice to model a system's behaviour via the use of multiple independent semi-parametric models. I presented two useful techniques for the modelling of computer programs execution. First I showed how to perform inference on brief runtime measurements, which often exhibit a long tail in their distribution. I then presented a novel treed Gaussian process data structure with better computational complexity than traditional Gaussian processes. Treed Gaussian processes can be used within Probabilistic-C++ as a non-parametric model to perform inference on large datasets.

- In **Chapter 5**, I presented BOAT's optimisation scheduling abstraction. The goal of this abstraction is to enable the easy and structured use of optimisations within optimisations. This has shown to be a useful technique in a number of contexts. In particular, I showed how decompositions, a known set of techniques in numerical optimisation, could be implemented in this way. I demonstrated, through the use of examples from the case studies, how decomposition methods could be used in the context of the numerical optimisation stage of a Bayesian optimisation.

- **Chapter 6** combined the techniques of Chapters 5 and 6 for the design of bespoke auto-tuners. I presented a method, based on Thompson sampling, to perform exploratory experiments while still being able to leverage decompositions. I also showed how one may be able to use BOAT's optimisation abstraction to leverage informative experiments that are cheaper than the objective function. Finally, I discussed a methodology to design an auto-tuner. I showed how, starting from a simple auto-tuner, one can diagnose the issues harming convergence, and listed the different ways in which these issues could be tackled.

- In **Chapter 7**, I discussed three case studies in which I applied BOAT to design bespoke auto-tuners: *(i)* A garbage collection case study in which I optimised the garbage configuration flags of a database application, *(ii)* a sort case study in which I tuned the parameters of decision trees which dynamically picked an efficient implementation of `std::sort` based on the input array's sortedness, and *(iii)* a neural network scheduling case study in which I optimised the distributed scheduling of a neural network onto a heterogeneous cluster. In each case study, I demonstrated that simple "one-size-fits-all" configurations yielded sub-optimal performance. I then showed that bespoke auto-tuners had better convergence than off-the-shelf frameworks. Finally, I discussed one of the benefits of using auto-tuners with structured Bayesian optimisation: the developer is able to inspect the learned model after the optimisation to diagnose the underlying system behaviour. In a way, the learned model can be seen as a high level profiler of the computation.

The contributions of this dissertation collectively serve to demonstrate the hypothesis stated in Chapter 1. First, that optimisations problems can benefit from leveraging a small amount of domain specific information. The abstractions offered by BOAT, presented in Chapters 3-6, allowed me to design multiple auto-tuners which significantly outperforms their off-the-shelf counterparts. Second, that these contributions make auto-tuning applicable in previously unexplored contexts. In particular, BOAT allowed me to apply auto-tuning to the distributed scheduling of neural networks, a context in which auto-tuning was previously intractable.

Future work

The abstractions introduced by BOAT offer a structured approach to designing a bespoke auto-tuner. There remains, however, many open questions. I discuss three:

> *How should we construct auto-tuners for the vertical optimisation of computer systems?*

In many real-world contexts, systems are used in a stack where each system uses the abstraction offered by the system below. In the case studies presented in this dissertation, I optimised a single application's performance. Ideally, however, each item in the stack would include its own bespoke auto-tuner, and BOAT's optimisation abstraction would be employed – in the context of a concrete application – to optimise the entire stack at once.

For example, consider the garbage collection case study in which I tuned the JVM garbage collection flags of a database application. There were two separate parts of my corresponding model. One predicting the JVM behaviour in terms of the rate and duration of garbage collection, and the other predicting the database's behaviour in terms of its latency. Ideally, both models would be developed independently so that the garbage collection models could be used to optimise any JVM application. How to design auto-tuners that offer the right abstraction, both in terms of their probabilistic model and the numerical optimisation of their parameters, is an open area of research. A context in which this could be especially interesting is the optimisation of real-time statistical applications which can trade-off accuracy for computations.

> *Should we consider the design of new, more flexible abstractions for systems which will then vastly adapt their implementation to the application?*

Although current systems have configurations parameters, their abstractions were designed with performance portability in mind. In the presence of high performance auto-tuners, we should consider whether more flexible and practical systems could be constructed, allowing for a vast range of underlying executions. As noted in Chapter 8, a bespoke auto-tuner strongly resembles an optimising compiler. Both attempt to maximise the performance of a computation. Hence, we could imagine a generic system which compiles complex applications to high performance executions.

Can some of the techniques presented in this dissertation be used for real-time optimisations?

In this dissertation, I have worked under the assumption that we were performing an iterative optimisation, and hence were able to measure the objective function multiple times before converging on a configuration. However, models of computations are also used to make real-time decisions, such as to plan the execution of database queries. In this case, the budget for the optimisation is lower than a single evaluation of the objective function. One could consider using some of the tools presented in this dissertation, such as Probabilistic-C++ or the use of decompositions, for the real-time scheduling of incoming computations.

This raises a new question: how should exploration be performed at runtime? The offline aspect of BOAT means that it is acceptable to perform exploratory experiments with possibly poor performance. To be applicable at runtime, new exploration strategies would have to be designed to guarantee a minimum quality of service.

Bibliography

[AAB$^+$00] Yair Amir, Baruch Awerbuch, Amnon Barak, Sean R. Borgstrom, and Arie Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):760–768, 2000. Cited on page 159.

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. Cited on page 139.

[AB13] Veronika Abramova and Jorge Bernardino. NoSQL databases: MongoDB vs Cassandra. In *Proceedings of the ACM International C* Conference on Computer Science and Software Engineering (C3S2E)*, 2013. Cited on page 132.

[ABC$^+$06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2006. Cited on page 158.

[ACW$^+$09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009. Cited on pages 37, 134, and 156.

[ADH10] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010. Cited on page 56.

[AKV$^+$14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: an extensible framework for program autotuning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation (PACT)*, 2014. Cited on pages 131, 134, 137, 147, and 156.

[Ama] Amazon. Amazon EC2 Instance Configuration. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-ec2-config.html`. Accessed: 2016-09-27. Cited on page 149.

[Apa16] Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org, 2016. Cited on pages 36 and 132.

[BACD97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 1997. Cited on page 157.

[BDD$^+$01] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001. Cited on page 157.

[BGO$^+$16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14:70–93, 2016. Cited on page 18.

[BKK$^+$98] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998. Cited on page 158.

[Boo16] Boost. Boost C++ Libraries. `http://www.boost.org/`, 2016. Last accessed 2016-09-19. Cited on pages 60, 67, and 91.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. Cited on pages 30 and 78.

[BXMM07] Stephen Boyd, Lin Xiao, Almir Mutapcic, and Jacob Mattingley. Notes on decomposition methods. *Notes for EE364B, Stanford University*, pages 1–36, 2007. Cited on page 98.

[cCH02] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2002. Cited on page 160.

[CCMGB06] Antonio J Conejo, Enrique Castillo, Roberto Mínguez, and Raquel García-Bertrand. *Decomposition techniques in mathematical programming: engineering and science applications*. Springer Science & Business Media, 2006. Cited on page 98.

[CF14] Yanshuai Cao and David J Fleet. Generalized product of experts for automatic and principled fusion of Gaussian process predictions. *arXiv preprint arXiv:1410.7827*, 2014. Cited on page 28.

[CGH$^+$16] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2016. Cited on pages 50, 51, 58, and 161.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1998. Cited on page 158.

[Chi] Soumith Chintala. Deepmark benchmark. `https://github.com/DeepMark/deepmark`. Cited on page 145.

[CJL$^+$08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, August 2008. Cited on page 160.

[CKF11] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011. Cited on page 139.

[CLCD07] Mung Chiang, Steven H Low, A Robert Calderbank, and John C Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, 2007. Cited on page 161.

[CLL$^+$15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. Cited on page 139.

[CLM16] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016. Cited on page 158.

[CMBJ16] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016. Cited on page 140.

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007. Cited on page 161.

[Coc70] John Cocke. Global common subexpression elimination. In *ACM Sigplan Notices*, volume 5, pages 20–24. ACM, 1970. Cited on page 158.

[Con63] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963. Cited on page 91.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010. Cited on pages 36 and 132.

[CT11] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011. Cited on page 158.

[DAV+15] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015. Cited on page 156.

[DCM+12] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012. Cited on page 139.

[DDFG01] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001. Cited on pages 51 and 58.

[DDK$^+$16] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. ProteusTM: Abstraction meets performance in transactional memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016. Cited on page 159.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008. Cited on pages 158 and 160.

[DJ09] Arnaud Doucet and Adam M Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704), 2009. Cited on page 51.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. Cited on page 159.

[DK14] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. Cited on page 159.

[DN15] Marc Peter Deisenroth and Jun Wei Ng. Distributed Gaussian processes. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015. Cited on pages 28 and 78.

[DR14] Nuno Diegues and Paolo Romano. Self-tuning Intel transactional synchronization extensions. In *Proceedings of the ACM/IEEE/USENIX International Conference on Autonomic Computing (ICAC)*, June 2014. Cited on page 161.

[DSH13] G. E. Dahl, T. N. Sainath, and G. E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013. Cited on page 33.

[DSH15] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015. Cited on page 123.

[DSY16] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Tuning the scheduling of distributed stochastic gradient descent with Bayesian optimization. In *NIPS Workshop on Bayesian Optimization*, December 2016. Cited on page 21.

[DSY17] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the ACM International Conference on World Wide Web (WWW)*, April 2017. To appear. Cited on page 21.

[DTB09] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009. Cited on page 157.

[EWA11] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart data structures: An online machine learning approach to multicore data structures. In *Proceedings of the ACM/IEEE/USENIX International Conference on Autonomic Computing (ICAC)*, 2011. Cited on page 161.

[FBK$^+$12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2012. Cited on page 160.

[FJ98] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1998. Cited on page 157.

[GES$^+$09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009. Cited on page 160.

[GGXS$^+$12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012. Cited on page 157.

[Gha15] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015. Cited on page 106.

[GHNR14] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE, FOSE track)*, 2014. Cited on page 48.

[GJ$^+$10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. Cited on pages 70 and 75.

[GL07] Robert B. Gramacy and Herbert K. H. Lee. Bayesian treed Gaussian process models with an application to computer modeling. *Journal of the American Statistical Association*, 2007. Cited on pages 29 and 79.

[GLG$^+$12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012. Cited on page 160.

[GMR$^+$08] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2008. Cited on pages 50, 51, and 161.

[GPF06] Andreas Gal, Christian W Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the ACM International Conference on Virtual Execution Environments (VEE)*, 2006. Cited on page 160.

[GSG$^+$16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016. Cited on page 159.

[Had] Apache Hadoop. Hadoop fair scheduler. `http://hadoop.apache.org/docs/stable1/fair_scheduler.html`. Cited on page 158.

[HBdF11] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for Bayesian optimization. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011. Cited on pages 32 and 119.

[HES$^+$10] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the ACM/IEEE/USENIX International Conference on Autonomic Computing (ICAC)*, 2010. Cited on page 160.

[HHK$^+$12] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E Miller, Sabrina M Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, et al. Self-aware computing in the Angstrom processor. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2012. Cited on page 160.

[HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization (LION)*. Springer, 2011. Cited on page 30.

[HLHG14] José Miguel Hernández-Lobato, Matthew W. Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In *Advances in Neural Information Processing Systems (NIPS)*, 2014. Cited on pages 31, 32, 82, and 115.

[HLL$^+$11] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011. Cited on page 158.

[HNADAD15] Jun He, Duy Nguyen, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Reducing file system tail latencies with Chopper. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 15)*, 2015. Cited on page 157.

[HO01] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001. Cited on pages 30, 92, and 94.

[HS12] Philipp Hennig and Christian J Schuler. Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13(Jun):1809–1837, 2012. Cited on page 32.

[HSC$^+$11] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011. Cited on page 160.

[IMMC08] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2008. Cited on page 161.

[IPC+09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2009. Cited on page 159.

[IY01] Eun-Jin Im and Katherine Yelick. *Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY*, pages 127–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Cited on page 157.

[JJNH91] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991. Cited on page 28.

[Joh14] Steven G Johnson. The NLopt nonlinear-optimization package, 2014. Cited on page 92.

[Jon01] Donald R. Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, 21(4):345–383, 2001. Cited on page 117.

[JPS93] Donald R Jones, Cary D Perttunen, and Bruce E Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993. Cited on pages 30, 92, and 94.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia (MM)*, 2014. Cited on page 139.

[KA01] Ken Kennedy and John R Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001. Cited on page 158.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques.* MIT press, 2009. Cited on page 48.

[KGE+15] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Performance evaluation of NoSQL databases: A case study. In *Proceedings of the Workshop on Performance Analysis of Big Data Systems (PABS)*, 2015. Cited on page 132.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983. Cited on pages 92, 93, and 94.

[KMN$^+$16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016. Cited on page 139.

[KMT98] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998. Cited on page 161.

[Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974. Cited on page 125.

[Kri51] Daniel G. Krige. A Statistical Approach to Some Basic Mine Valuation Problems on the Witwatersrand. *Journal of the Chemical, Metallurgical and Mining Society of South Africa*, 52(6):119–139, December 1951. Cited on page 23.

[Kri14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014. Cited on pages 139 and 145.

[LCC06] Jang-Won Lee, Mung Chiang, and A Robert Calderbank. Utility-optimal medium access control: Reverse and forward engineering. In *INFOCOM*, 2006. Cited on page 161.

[MAB$^+$10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010. Cited on page 160.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms.* Cambridge University Press, 2003. Cited on page 51.

[MAM10] Iain Murray, Ryan Prescott Adams, and David J.C. MacKay. Elliptical slice sampling. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010. Cited on page 27.

[Mat62] Georges Matheron. Traité de géostatistique appliquée. *Editions Technip, Paris*, 14, 1962. Cited on page 23.

[MGJ13] Alberto Magni, Dominik Grewe, and Nick Johnson. Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the ACM Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*, 2013. Cited on page 157.

[MJ01] Dragan Mirković and S. Lennart Johnsson. Automatic performance tuning in the UHFFT library. In *Proceedings of the International Conference on Computational Science (ICCS)*, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. Cited on page 157.

[MO06] Edward Meeds and Simon Osindero. An alternative infinite mixture of Gaussian process experts. *Advances in Neural Information Processing Systems (NIPS)*, 2006. Cited on page 28.

[MSP14] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014. Cited on pages 51, 58, 61, and 161.

[MTŽ78] J. Močkus, V. Tiesis, and A. Žilinskas. The application of Bayesian methods for seeking the extremum. *Toward Global Optimization*, 2:117–128, 1978. Cited on pages 29 and 31.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* The MIT Press, 2012. Cited on pages 33, 65, 67, 105, 158, and 159.

[MWG$^+$] T Minka, J Winn, J Guiver, S Webster, Y Zaykov, B Yangel, A Spengler, and J Bronskill. Infer.NET 2.6, 2014. *Microsoft Research Cambridge.* Cited on pages 51 and 161.

[ND14] Jun Wei Ng and Marc Peter Deisenroth. Hierarchical mixture-of-experts model for large-scale Gaussian process regression. *arXiv preprint arXiv:1412.3078*, 2014. Cited on page 28.

[Nea94] R. M Neal. *Bayesian Learning for Neural Networks.* PhD thesis, Dept. of Computer Science, University of Toronto, 1994. Cited on page 23.

[Nea11] Radford M Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2:113–162, 2011. Cited on pages 27 and 161.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965. Cited on page 156.

[NM09] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009. Cited on page 157.

[O'H78] A. O'Hagan. Curve Fitting and Optimal Design for Prediction (with discussion). *Journal of the Royal Statistical Society, Series B*, 40(1):1–42, 1978. Cited on page 69.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120. Cited on page 160.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990. Cited on page 158.

[Plu03] Martyn Plummer. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the International Workshop on Distributed Statistical Computing*, volume 124, page 125, 2003. Cited on page 161.

[PMS$^+$04] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications (IJHPCA)*, 18(1):21–45, 2004. Cited on page 157.

[Puk93] Friedrich Pukelsheim. *Optimal design of experiments*, volume 50. siam, 1993. Cited on page 159.

[PW14] Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2014. Cited on pages 55, 56, 58, and 161.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. Cited on page 105.

[Ras06] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006. Cited on pages 23 and 25.

[RBMZ15] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015. Cited on page 160.

[RG01] Carl Edward Rasmussen and Zoubin Ghahramani. Infinite mixtures of Gaussian process experts. In *Advances in Neural Information Processing Systems (NIPS)*, 2001. Cited on pages 28 and 78.

[RG15] Razvan Ranca and Zoubin Ghahramani. Slice sampling for probabilistic programming. *arXiv preprint arXiv:1501.04684*, 2015. Cited on pages 61 and 161.

[RGVS$^+$12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, August 2012. Cited on page 132.

[RMZ13] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013. Cited on page 160.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2011. Cited on page 140.

[Sco10] Steven L Scott. A modern Bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010. Cited on page 30.

[SDS$^+$13] Kevin Swersky, David Duvenaud, Jasper Snoek, Frank Hutter, and Michael Osborne. Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces. In *NIPS workshop on Bayesian Optimization*, 2013. Cited on page 40.

[SFD$^+$14] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *INTERSPEECH*, 2014. Cited on page 145.

[SFLD15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, New York, NY, USA, 2015. ACM. Cited on page 157.

[SG05] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian processes using pseudo-inputs. In *Advances in Neural Information Processing Systems (NIPS)*, 2005. Cited on pages 28 and 78.

[SKSK10] Niranjan Srinivas, Andreas Krause, Matthias Seeger, and Sham M. Kakade. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010. Cited on pages 32, 83, and 119.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan Prescott Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, 2012. Cited on pages 28, 33, 36, 131, 134, 137, and 147.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. Cited on page 145.

[SRS+15] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, and Ryan P. Adams. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, 2015. Cited on page 30.

[SSA14] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw Bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014. Cited on page 123.

[SSW+16] 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. Cited on pages 30 and 33.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14:1–14:42, May 2009. Cited on page 161.

[STB+96] David J Spiegelhalter, Andrew Thomas, Nicky G Best, Wally Gilks, and D Lunn. BUGS: Bayesian inference using Gibbs sampling. *Version 0.5,(version ii) http://www. mrc-bsu. cam. ac. uk/bugs*, 19, 1996. Cited on pages 50, 51, and 161.

[SWH+14] Bobak Shahriari, Ziyu Wang, Matthew W. Hoffman, Alexandre Bouchard-Côté, and Nando de Freitas. An Entropy Search Portfolio for Bayesian Optimization. *arXiv*, 1406.4625, 2014. Cited on pages 32 and 119.

[SWL03] Matthias Seeger, Christopher Williams, and Neil Lawrence. Fast forward selection to speed up sparse Gaussian process regression. In *Artificial Intelligence and Statistics 9*, number EPFL-CONF-161318, 2003. Cited on pages 28 and 78.

[SZL+14] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. MRTuner: a toolkit to enable holistic optimization for Mapreduce jobs.

Proceedings of the VLDB Endowment, 7(13):1319–1330, 2014. Cited on page 158.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. Cited on page 139.

[Tho33] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933. Cited on pages 30 and 109.

[Tit09] Michalis K Titsias. Variational learning of inducing variables in sparse Gaussian processes. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2009. Cited on pages 28 and 78.

[TKD$^+$16] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016. Cited on pages 61 and 161.

[TvdMW15] David Tolpin, Jan-Willem van de Meent, and Frank Wood. Probabilistic programming in Anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*. Springer, 2015. Cited on pages 51, 58, 61, and 161.

[VCC11] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the ACM/IEEE/USENIX International Conference on Autonomic Computing (ICAC)*, 2011. Cited on page 160.

[VDY05] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005. Cited on page 157.

[VPK$^+$15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2015. Cited on page 159.

[VVW09] Julien Villemonteix, Emmanuel Vazquez, and Eric Walter. An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*, 44(4):509–534, 2009. Cited on page 32.

[VYF^{+}16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016. Cited on page 159.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. Cited on page 106.

[WD98] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC)*, 1998. Cited on page 157.

[WJLH06] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, December 2006. Cited on page 161.

[WvdMM14] Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2014. Cited on page 161.

[YAK14] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 26:1–26:14, 2014. Cited on page 158.

[YL96] Frank Yellin and Tim Lindholm. The Java virtual machine specification. *Addison-Wesley*, 1996. Cited on pages 36 and 132.

[YN09] Chao Yuan and Claus Neubauer. Variational mixture of Gaussian process experts. In *Advances in Neural Information Processing Systems (NIPS)*, 2009. Cited on page 28.

[ZLC10] Jingren Zhou, P-A Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2010. Cited on page 158.