THE UNIVERSITY
of EDINBURGH

Optimisation and Bayesian Optimality

Thomas Joyce

Doctor of Philosophy

Institute of Perception, Action and Behaviour

School of Informatics

University of Edinburgh

2016

Abstract

This doctoral thesis will present the results of work into optimisation algorithms. We first give a detailed exploration of the problems involved in comparing optimisation algorithms. In particular we provide extensions and refinements to no free lunch results, exploring algorithms with arbitrary stopping conditions, optimisation under restricted metrics, parallel computing and free lunches, and head-to-head minimax behaviour. We also characterise no free lunch results in terms of order statistics.

We then ask what really constitutes understanding of an optimisation algorithm. We argue that one central part of understanding an optimiser is knowing its Bayesian prior and cost function. We then pursue a general Bayesian framing of optimisation, and prove that this Bayesian perspective is applicable to all optimisers, and that even seemingly non-Bayesian optimisers can be understood in this way. Specifically we prove that arbitrary optimisation algorithms can be represented as a prior and a cost function. We examine the relationship between the Kolmogorov complexity of the optimiser and the Kolmogorov complexity of it's corresponding prior. We also extended our results from deterministic optimisers to stochastic optimisers and forgetful optimisers, and we show that uniform randomly selecting a prior is not equivalent to uniform randomly selecting an optimisation behaviour.

Lastly we consider what the best way to go about gaining a Bayesian understanding of real optimisation algorithms is. We use the developed Bayesian framework to explore the affects of some common approaches to constructing meta-heuristic optimisation algorithms, such as on-line parameter adaptation. We conclude by exploring an approach to uncovering the probabilistic beliefs of optimisers with a "shattering" method.

Lay Summary

In science and engineering it is common to encounter situations in which you want to find the best solution from a very large number of possible solutions. When the number of possible solutions is too large for it to be feasible to simply try them all, then one must instead rely on some strategy for finding a satisfactory solution. There are many such strategies, and in this thesis we look at a subset of these strategies called meta-heuristic optimisers. Meta-heuristics optimisers can be though of as simple, broadly applicable strategies for finding good solutions.

Firstly we examine the ways in which these optimisers can be compared. We see that there are important senses in which all meta-heuristic optimisers are equal, but that there are also situations which result in certain meta-heuristics being provably better than others.

We then argue that an interesting way to explore meta-heuristic optimisers is to rephrase them in terms of probabilistic beliefs. Moreover, we show that such a rephrasing is always possible. Roughly speaking, we show that any simple strategy for searching for good solutions is optimal behaviour given certain beliefs about the sorts of problems likely to be encountered.

Lastly we look at how one might go about actually uncovering the prior beliefs that result in a particular meta-heuristic behaviour, and we introduce a simple numerical approach and explore it's results.

Acknowledgements

Many thanks to my family and friends. Also thanks to all the doctors, nurses, priests, magic-users and nutritionists who helped me get where I am today. Thanks also to my second and third supervisors, Taku Komura and Subramanian Ramamoorthy, for keeping me on the track. Finally, a huge and resonating thanks to my primary supervisor, Michael Herrmann. Smarter kinder guys are hard to come by.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Thomas Joyce*)

Table of Contents

Chapter 1

Introduction and Preliminaries

1.1 Overview

This thesis is about optimisation algorithms. In particular, it is about how we might better understand the behaviour of optimisation algorithms. Nowadays there are a variety of powerful optimisers in regular use, but we still do not fully understand how many of these algorithms actually work.

The task of improving our understanding of optimisers can be approached from several directions. For example, one can endeavour to achieve a theoretically rigorous understanding of a particular algorithm, search for the best optimisers for some particularly important problem type, or look at the behaviour of a commonly used optimiser in practice. All of these are valuable avenues of research. However, in this thesis we try to get a better understanding of meta-heuristic optimisers in a broad sense.

Optimisation algorithms are, in general, difficult to understand. There are a vast number of optimisation algorithms in the literature, and any well established optimiser has hundreds of proposed variants claiming to result in improved behaviour. Moreover, even if we focus only on broad classes of algorithms there are still many competing, and often overlapping approaches: swarm algorithms, genetic algorithms, Gaussian process optimisation, gradient methods, and simulated annealing to name but a few. This diversity makes a general framework for understanding optimisation difficult.

In fact, often it isn't even clear what "understanding" an optimisation algorithm means. What sort of understanding do we want to achieve, and to what end? In many ways we already do understand optimisation algorithms. After all, researchers devised and implemented them, and others have refined and improved on these originals. Yet although the heuristic idea and low level implementation are generally both well under-

stood, the behaviour that emerges when the algorithm is actually run on real problems is much less well understood. This is especially true when we are interested in optimisers for functions with multiple optima and complex high-dimensional structure.

In this work we focus on the *behaviour* of optimisers removed from any particular implementation. Given the diversity of optimisation algorithms it is not obvious whether there exists a simple unified way of understanding them. Can all optimisation algorithms be described in some unified framework? We answer this question in the positive and show that any optimisation algorithm can be understood as Bayesian optimal behaviour with respect to some prior and cost function.

In a Bayesian optimisation setting the prior is a probability distribution which describes prior beliefs about the sorts of problem that the optimiser will encounter. Equivalently it can be seen as characterising uncertainty about the functions that will be optimised. When optimising a particular function, data from sampling is used to update the beliefs, and reduce uncertainty. The cost function is then used to decide what to do base on the current beliefs (in optimisation this is just the decision of where to sample next). In this way a prior and a cost function fully determine a Bayesian optimiser's behaviour.

We will show that actually, any optimisation algorithm can be represented as a prior and a cost function, not just those normally considered to be Bayesian. For example, the heuristic-based particle swarm optimisation algorithm can be understood as being Bayesian. In fact, we will show that there are some cost functions which can potentially produce any optimisation behaviour. For these cost functions different behaviours simply correspond to different priors. As a result, all optimisation algorithms can be thought of as the result of particular prior beliefs. Any optimiser can be expressed just as a prior.

In this thesis we argue that an important part of understanding an optimisation algorithm's behaviour is exactly being able to phrase it in a Bayesian way. As just discussed, we show that this is always possible. We also show that this representation of the algorithm is compact. Further we show that this representation is agnostic with regard to the algorithm's aims; different interpretations of what the algorithm is *trying to do* produce different but behaviourally equivalent formulations of the optimiser.

For example, if we decide the optimisation algorithm always samples the expected minimum then one formulation results, if instead we believe the optimisation algorithm is always trying to gain maximum information about the location of the minimum then we get an different, but equivalent, formulation of the optimiser. In other words, for

any optimiser using a cost function that makes it always sample the expected minimum, there is a different algorithm that uses a maximum information gain cost function but produces exactly the same behaviour.

The work in this thesis then can be summarised as an effort to understand optimisation in a rigorous and unified way, and we hope that it is at least a step toward that goal.

1.2 Problem Statement

Optimisation problems are very varied, and so too are our approaches to solving them. Some optimisation scenarios are very well understood, for example explicitly solving a low degree polynomial to find extrema, while some problems are still essentially beyond our capabilities, for example minimising very high dimensional, slow to evaluate problems with missing, non-stationary or uncertain data and no gradient information. In particular, search and optimisation problems tend to be especially difficult if they:

- are high-dimensional.
- have no gradient information.
- are expensive to evaluate.
- have missing, non-stationary or uncertain data.

When problems are sufficiently simple, we are now able to efficiently solve them with well understood techniques. However, many important problems are either still essentially insoluble or are best addressed by relatively new methods for which full theoretical understanding is still required. These sorts of problems tend to fall in the categories in the above list.

Meta-heuristics are broadly used for the difficult problems in this latter category, and many of the more popular meta-heuristic optimisers are not yet rigorously characterized theoretically. Of course, some are relatively well understood, but when this is the case it generally came about though a detailed and specialized investigation into the particularities of the algorithm concerned. In other investigations simplified versions of powerful optimisers have been studied, resulting in a distinction between the simplified version of the optimiser that we understand, and the version actually used in

reality, which we do not. Further we still lack a generally applicable framework for developing understanding of optimisers, and for comparing and contrasting the different meta-heuristic approaches.

This problem is made more difficult by the no free lunch theorems we will cover in the next chapter. Essentially they show that all optimisers are equal on average, and thus it is not a matter of determining whether or not an optimisation algorithm is good, and algorithm evaluation can not be a simple case of pass or fail, but rather we have to tease out the problems that a given optimiser is particularly suited to.

Thus it is clear that there are gaps in our understanding of optimisation. On the one hand there are optimisation algorithms that are well understood behaviourally. These optimisers tend to either be a very simple procedure, such as grid search, or be in the family of explicitly Bayesian optimisation algorithms.

On the other hand, there are many optimisation algorithms that seem to be neither extremely simple nor Bayesian in their behaviour. In fact, these algorithms are often very popular in practice, generally producing quality solutions quickly and reliably. For example, common optimisation algorithms like gradient descent and simulated annealing, and naturally inspired optimisers like genetic algorithms, particle swarm and cuckoo search do not generally make any reference to a probability distribution over the space of possible functions seemingly required in a Bayesian framework, they seem not to work with prior beliefs that are updated in accordance with Bayes' rule over the course of the optimisation process. At the same time they are capable of complex dynamic behaviour that is far from straight forward to analyse and understand.

Put simply, there is a void between pragmatic meta-heuristic algorithms and theoretically rigorous approaches to optimisation. The problem, then, is to understand the spectrum of optimisation algorithms in some unifying way. A framework for understanding is only useful if it allows us to think more clearly about an area, or naturally yields novel insights. In particular, it should enable us to address various questions: What are the key defining features of an optimiser? What makes optimisers similar? What makes optimisers good?

1.3 Scope and Limitations

This work focuses on optimisation of functions mapping between finite sets. We do not consider extension to the case of continuous problems, nor explicitly consider multi-objective optimisation.

We review and extend result on optimisation algorithm comparison, in particular no free lunch results. We conclude that, as argued elsewhere, what determines an optimisers effectiveness is it's alignment with the problem at hand.

We then argue that a Bayesian interpretation is the right way to understand this idea of alignment, and moreover, that a Bayesian framing is possible for any optimisation algorithm. We prove this for on-policy deterministic optimisers and then we look at how the same ideas can be extended to full behaviours and stochastic optimisers.

Although we show that priors necessarily exist for all optimisers, we do not present general methods for extracting these priors from algorithm implementations. Such a task is certainly difficult, and perhaps impossible in general.

We briefly explore the effect on the prior beliefs of combining optimisers, but a full understanding of how hybrid optimisers priors relate to the priors of their constituent parts isn't reached. In fact, there is the potential for something like an algebra of optimisers, where algorithm behaviours could be added and subtracted, resulting in new behaviours. This idea is introduced as a possible application of the Bayesian framing, but not developed here.

1.4 Roadmap

Here we provide details of the layout of the thesis. The remainder of this chapter is given over to preliminaries, in particular we define the terms and symbols we use throughout, and state and prove some central underlying results from which the thesis builds.

Chapter two consists of both a literature review and a detailed exploration of the problems involved in comparing optimisation algorithms. In particular it looks at benchmarking and the various no free lunch results, which are a number of results showing that in a broad range of situations no algorithm can be expected to achieve better results than any other. These results will be shown to be central in motivating our later chapters.

Chapter three introduces the concept of Bayesian optimisation, and starts the argument that a Bayesian perspective is applicable to all optimisation algorithms. In particular in the chapter we prove that deterministic non-resampling optimisers are all Bayesian optimal with respect to some cost function and probability distribution over problem functions. In this chapter we also discuss Kolmogorov complexity, which can be thought of as a rigours measure of an algorithm's complexity in a theoretically im-

portant sense. In particular we give a proof of a relationship between the Kolmogorov complexity of an optimisation algorithm's implied prior beliefs and the Kolmogorov complexity of the algorithm itself.

In chapter four we look at how the approach to understanding optimisers can be extended to stochastic optimisers and forgetful optimisers. We also explore its application to some general approaches to constructing meta-heuristic optimisation algorithms, like algorithm switching, and on-line parameter adaptation.

In chapter five we look to understand some popular meta-heuristic optimisers, and try to tease out their underlying beliefs. Here we apply the ideas developed in the previous chapters.

Finally, in chapter six we conclude, look more clearly at some limitations of the work, and suggest future avenues for research.

1.5 Novel Contributions

This thesis contains several results and generalisations which are, to the best of our knowledge, new. The main new results are listed here.

- Chapter 2 contains several no free lunch (NFL) extensions and refinements:
  - NFL for algorithms with arbitrary stopping conditions (2.5.2).
  - NFL for off-policy behaviour (2.6.6).
  - NFL for restricted metrics (2.6.7).
  - NFL and parallelism (2.7.7).
  - NFL in minimax behaviour (2.8.7).
  - NFL order statistics and the curse of dimensionality (2.9.9).
- Chapter 3 looks at representing optimiser in a Bayesian way:
  - All deterministic on-policy behaviour is Bayesian (3.5).
  - Different cost functions produce different prior distributions (3.8).
  - Admissible cost functions exist (3.8).
  - The Kolmogorov complexity of the implied prior is related to the Kolmogorov complexity of the optimisation algorithm (3.7).

- Chapter 4 extends the Bayesian representation to a wider range of optimisers:
  - Uniform randomly selecting a prior is not uniform randomly selecting a behaviour (4.2.2.1).
  - Full behaviours can be represented as prior and cost function pairs where the prior is conditioned on the starting data (4.3.2).
  - Forgetful optimisers are Bayesian (4.4).
  - Stochastic optimisers can be strictly worse behaviourally than deterministic optimisers, and can never be behaviourally better (4.5.5).
  - Hybrid optimisation algorithms do not straightforwardly combine the priors of the constituent optimisers (4.6.1).
- Chapter 5 examines some prominent optimisers using the framework:
  - Algorithm shattering proof of concept (5.2).

1.6 Preliminaries

This thesis is concerned with optimisation algorithms. Optimisation algorithms are procedures for solving optimisation problems. An optimisation problem is a situation in which we want to find the best possible solution from a set of feasible solutions.

In the following sections we will formally define the concepts required to discuss optimisation, and provide examples. Definitions of functions, data, sampling policies and optimisation algorithms will be introduced in that order. The definitions given in this chapter are just the formalising of some general concepts and specification of notation. We also use some combinatorics to calculate the number of unique optimisation algorithms, prove some basic results, and generally lay the ground for the main body of the thesis.

Readers familiar with optimisation may wish to skip this section, barring perhaps subsection 1.7.2 where the way we represent partially observed functions is detailed, and Theorem 1, which is used extensively in the next chapter.

1.7 Functions and Data

We now start with our most fundamental definitions. Functions are used as formal representations of the problems we want to solve, and data allows us to represent partial

knowledge of functions.

1.7.1 Functions

A function is a mapping from a set X to a set Y which we write as $f : X \to Y$ where f is the name of the function. $f : X \to Y$ can be thought of as a rule for assigning to each element in X and element in Y. We call X the domain and Y the range (elsewhere Y is sometimes called the co-domain).

For the majority of this thesis we consider deterministic functions $f : X \to Y$, where X and Y are finite sets with $|X| = n$ and $|Y| = m$. We assume that neither X nor Y are empty, and thus $n \neq 0 \neq m$. We denote the set of all such functions $\mathcal{F}$, with $|\mathcal{F}| = m^n$.

As we need to represent functions frequently throughout the thesis we introduce a simple concise representation scheme. Without loss of generality assume that $X = \{1,2,\ldots,n\}$, then for any $f \in \mathcal{F}$ we can write an ordered list of y values $y_1, y_2, \ldots, y_n$ where $y_i = f(i)$. This ordered list of y values uniquely and fully describes f. When there is no ambiguity (e.g. in the case where $Y = \{0,1\}$) we will sometimes omit the commas between elements.

Example 1 (Function)**.** *Let $X = \{1,2,3\}$ and $Y = \{0,1\}$. We can then write out all $f \in \mathcal{F}$: 000, 001, 010, 011, 100, 101, 110, 111. For example, 110 corresponds to the case where $f(1) = 1$, $f(2) = 1$ and $f(3) = 0$. Also note that $|\mathcal{F}| = 8$ as $|Y|^{|X|} = m^n = 2^3 = 8$.*

In the optimisation literature there are many terms used to refer to the function being optimised. It is called variously the "cost function", "fitness function", "target function", "objective function", "problem" and even simply "the function". Here we use (cost) function.

We now introduce some extensions to the basic function.

Definition 1 (Bijection)**.** *A function $f : X \to Y$ is a bijection iff $\forall\, x_1, x_2 \in X$, $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$ and $\forall\, y \in Y$, $\exists\, x \in X$ such that $f(x) = y$.*

Definition 2 (Permutation)**.** *A permutation is a bijection from a set onto itself. That is, $\phi : X \to X$ is a permutation iff ϕ is a bijection.*

Note that by a counting argument, if $|X| = n$ then there are $n!$ unique bijections $\phi : X \to X$. The set of possible permutations of X can be thought of as the set of possible re-orderings of X.

Definition 3 (Function Permutation)**.** *Let ϕ be a permutation $\phi : X \to X$, and let $f : X \to Y$ be an arbitrary function. We call f_ϕ a function permutation where we define $f_\phi(x) = f(\phi(x))$.*

A function permutation can be thought of as re-ordering the output values. The resulting function has exactly the same outputs, but they now correspond to different inputs.

Example 2 (Function Permutation)**.** *Let $f : \{1,2,3,4,5\} \to \{1,2,3,4,5,6,7\}$ with $f = 2,5,2,3,7$ and let ϕ be a permutation $\phi = 2,5,4,1,3$, then $f_\phi = 5,7,3,2,2$. See Figure 1.1 for a graphical representation of this example.*

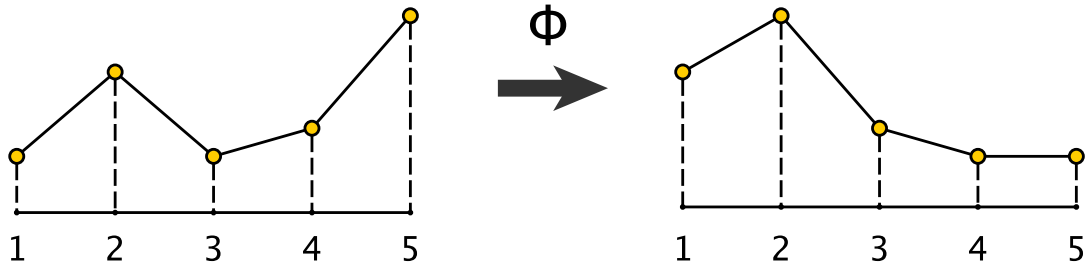

Figure 1.1: Function permutation example, $f = 2,5,2,3,7$ (left) is permuted by $\phi = 2,5,4,1,3$, resulting in $f_\phi = 5,7,3,2,2$ (right).

Definition 4 (CUP)**.** *Let G be a set of functions mapping X to Y. We say G is closed under permutation, or CUP, iff for any permutation $\phi : X \to X$, $f \in G \implies f_\phi \in G$.*

Definition 5 (Permutation Closure)**.** *Let G be a set of function mapping X to Y. We define G_{cup} as the smallest set containing G that is closed under permutation.*

Definition 6 (Multi-set)**.** *A multi set is a set in which elements can occur multiple times. Another way to think of a multi-set is as a set in which each element has an associated count.*

1.7.1.1 Binary Functions

In the examples and figures throughout this thesis we frequently use binary functions. However, as most optimisers work on the space of floating point numbers it is not immediately clear that binary functions are suitable. In fact we will see that for the

majority of results discussed in the coming chapters the range and domain of the functions concerned can be arbitrary finite sets, and thus the use of binary functions is just useful for brevity. When necessary we will make use of other ranges, but we will use $\{0,1\}$ as a default for simplicity.

1.7.2 Data

When discussing optimisation we frequently need to describe a situation in which only a limited amount is known about a function f. We now introduce a scheme for describing these partial knowledge situations. Note that we focus on noise-free optimisation, so, when we sample f at x we always get exactly $f(x)$ as an output.

Assume we know X and Y but only know some of the values $y = f(x)$. In this thesis we will represent such partial knowledge as a function $d : X \to Y \cup \{?\}$ which we call the data. The question mark "?" represents an x value for which the $f(x)$ value is unknown. For all x values where we know $y = f(x)$ let $d(x) = f(x)$, for all other values of x we let $d(x) = ?$. In other words, if for some $x \in X$ we have $d(x) = ?$ then we currently do not know $f(x)$. Let $\mathcal{D}$ be the set of all possible data for functions in $\mathcal{F}$. It follows that $|\mathcal{D}| = (m+1)^n$. We refer to the data where $\forall\, x \in X,\ d(x) = ?$ as "no data", as this represents the situation in which we know nothing about the cost function.

Example 3 (Data)**.** *Let $X = \{1,2,3,4\}$ and $Y = \{0,1\}$. Assume we have partial knowledge of some $f \in \mathcal{F}$. In particular we know that $f(2) = 1$ and $f(4) = 1$. We can represent this data as a function d where $d(1) = ?$, $d(2) = 1$, $d(3) = ?$ and $d(4) = 1$. As discussed in 1.7.1 his can be represented more succinctly as: ?1?1. Note that, assuming the data is accurate, there are four possible functions consistent with the data: 0101, 0111, 1101 and 1111.*

Another common way of formalising the idea of data is as a set of known X and Y value pairs (x,y). When $|X|$ is large and we do not know much about f this is a much more succinct representation than the method described above, and we will occasionally use it for this reason. Using this representation ?1?1 would be written as $\{(2,1),(4,1)\}$. Another benefit of this representation is it allows us to show if we have sampled the same point multiple times, e.g. the multi-set $\{(2,1),(4,1),(2,1),(2,1)\}$ is still ?1?1 but it can be seen that we sampled $x = 2$ three times (always returning the same y value as f is deterministic).

1.8 Sampling Policies and Optimisation Algorithms

We initially considering the set of *all* possible optimisation algorithm behaviours over functions $f : X \to Y$, unrestricted by theoretical limitation on what can feasibly be computed (in the lifetime of the universe for example) and unconcerned with implementability.

Optimisation algorithms are intended to solve optimisation problems. Generally this involves a sequential process: make a decision about where to sample the cost function, sample there and get some new information, make a decision about where to sample next, and so on. To this end we define a *sampling policy*, which is the decision making component of the optimisation algorithm in isolation.

1.8.1 Sampling Policy

The situation discussed throughout this thesis is the following: you have a function $f : X \to Y$ which you can evaluate at any $x \in X$. Your current knowledge of the f is represented by data d.

Intuitively, a sampling policy is a rule for deciding where to sample next, based on the current data; Given the current data it specifies which of the potentially numerous x we should evaluate next.

Definition 7 (Sampling Policy)**.** *A sampling policy s is a function $s : \mathcal{D} \to X$, where $\mathcal{D}$ is the set of all possible data, as defined in 1.7.2.*

Example 4 (Sampling Policy)**.** *Let $X = \{1,2\}$ and $Y = \{0,1\}$. An example sampling policy s is: $s(??) = 1$, $s(0?) = 2$, $s(1?) = 1$, $s(?0) = 1$, $s(?1) = 1$, $s(00) = 1$, $s(01) = 1$, $s(10) = 1$, $s(11) = 1$.*

Definition 8 (Non-Repeating Sampling Policy)**.** *A non-repeating sampling policy s is a function $s : \mathcal{D} \to X$ s.t. if $s(d) = x$ then either $d(x) = ?$ or $|d| = m$.*

Example 5 (Non-Repeating Sampling Policy)**.** *Let $X = \{1,2\}$ and $Y = \{0,1\}$. Then the set of possible data $\mathcal{D} = \{??, ?0, ?1, 0?, 1?, 10, 01, 00, 11\}$ and $s(??) = 1$, $s(?0) = 1$, $s(?1) = 1$, $s(0?) = 2$, $s(1?) = 2$, is a non-repeating sampling policy.*

We will mostly be considering non-repeating sampling policies, and when there is no ambiguity we will just call them sampling policies.

1.8.2 Optimisation Algorithms

So far we have defined sampling policies, which decide where to sample given data. However, we can add to our data by evaluating the cost function f at some x and updating d by setting $d(x) = f(x)$. Essentially, an optimisation algorithm is the repeated use of a sampling policy, adding to our data each time we make a sample, until a termination condition is reached. For the time being we fix our termination condition to: terminate iff we have sampled f at every $x \in X$. Given this fixed termination condition an optimisation algorithm is fully specified by a choice of sampling policy.

Definition 9 (Optimisation Algorithm)**.** *An optimisation algorithm A based on sampling policy s is iterated use of that sampling policy and data updating:*

1. *Set d to be the initial data (generally no data, but see Section 1.8.3.2).*
2. *If $? \notin d$ then terminate.*
3. *Sample at $x = s(d)$ and add the result to the data d, i.e. set $d(x) = f(x)$.*
4. *Go to step 2.*

1.8.3 Optimisation Algorithm Behaviour

One useful observation on the behaviour of deterministic optimisation algorithms is the existence of what can be thought of as on-policy and off-policy behaviour. This distinction is important for later proofs, and so we make it clear here.

1.8.3.1 On-Policy Behaviour

The on-policy behaviour of an algorithm is the sampling policy on a restricted set of possible data inputs. Specifically it is the behaviour on inputs potentially seen when the algorithm is run until termination starting with no information about f. The on-policy behaviour is generally not the full sampling policy, as there are data inputs that will never be seen in the normal execution of the algorithm, regardless off the function f being sampled. This is clarified in the following example:

Example 6 (On-Policy Behaviour)**.** *Consider the sampling policy described in example 5. $d(??) = 1$, and so an optimisation algorithm using this sampling policy will initially sample $f(1)$, and thus the data $?0$ and $?1$ will never be seen, as those data are the possible results from evaluating $f(2)$ first. It is not that $f(2)$ will never be evaluated, rather that if we follow the policy, $f(2)$ will never evaluated first.*

1.8.3.2 Off-Policy Behaviour

Off-policy behaviour is the sampling policy restricted to exactly those data inputs that do not appear in the on-policy behaviour. Thus together the off-policy and on-policy behaviours describe the full behaviour.

Example 7 (Off Policy Behaviour)**.** *Figure 1.2 shows the off-policy and on-policy behaviours for a sampling policy* $s : \mathcal{D} \to X$*, where* $X = \{1,2,3\}$ *and* $Y = \{0,1\}$*. The sampling policy* s *always samples the lowest (i.e. leftmost) unsampled* x*. For example* $s(?1?) = s(?0?) = 1$*,* $s(0??) = s(1?0) = 2$*.*

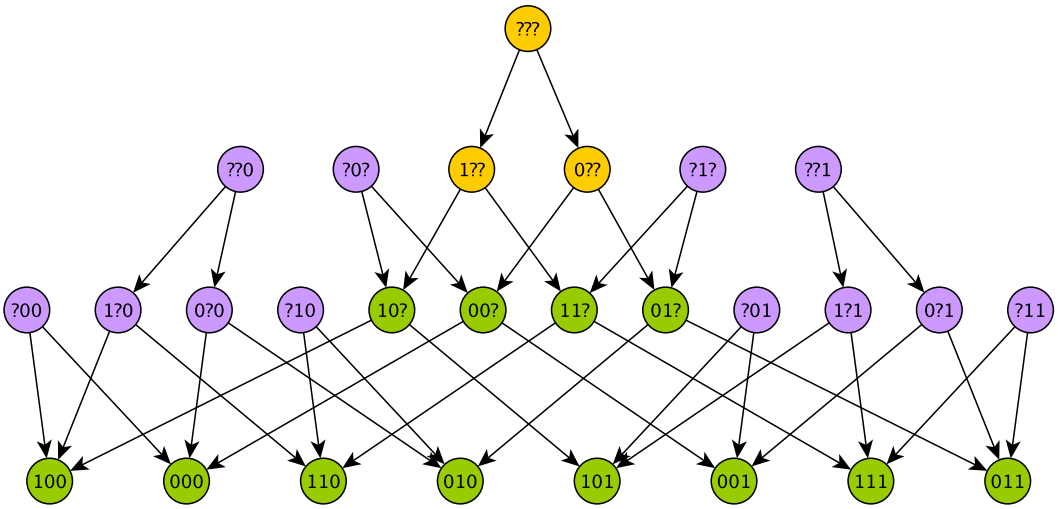

Figure 1.2: The full sampling policy behaviour graph. Yellow nodes are strictly on-policy, purple nodes are strictly off-policy, and green nodes are on-policy, but are also reachable from off-policy nodes.

Broadly speaking, the on-policy behaviour is all the behaviour that is possible in the normal running of the optimiser, and off-policy behaviour is that that results from starting the optimisation algorithm with some initial (off-policy) knowledge of the cost function.

1.8.4 Representing Optimisation Algorithm Behaviour

When discussing optimisation algorithms it is often helpful to consider their behaviour represented as a behaviour graph, which we now define:

Definition 10 (Behaviour Graph)**.** *An optimisation algorithm's behaviour can be represented as a directed graph, in which nodes represent possible data and the edges show all potential transitions between data resulting from sampling the cost function in accordance with the optimiser's sampling policy. We call such a graph the behaviour graph of the optimiser. See Figure 1.2 for an example of a behaviour graph.*

We now show that if we restrict attention to on-policy behaviour of deterministic optimisation algorithms, then the behaviour graph is in fact a tree. This is a key observation and is used variously throughout the rest of the thesis.

Theorem 1 (Optimisation Algorithm Tree Representation)**.** *A deterministic non-resampling optimisation algorithm's on-policy behaviour can be uniquely represented as a directed graph. In fact this directed graph is a balanced rooted tree.*

Proof. From the definition of on-policy behaviour we start the optimisation with no data about the function, which we represent as the root node of the tree. The algorithm's sampling policy determines which x should be sampled given no data, say x_1. When the algorithm performs the sample at x_1 there are $|Y| = m$ possible results, and each of these results necessarily leads to different data (as the data is just a description of the result). Thus, we can potentially transition to any of m new data nodes by appending the value of $(x_1, f(x_1))$ to the data.

At this point, either the node we are at represents data with no unknown cost function output values, in which case the algorithm halts and we are at a leaf, or the node contains at least one value for which $d(x) = ?$. In this second case the sampling policy will select one of these x to be sampled, and again m possible results exist, we follow one of the possible edges (dependent on the results of the sample) and then repeat the process until we do eventually reach a leaf.

We now show that all paths do eventually lead to a leaf. Whenever an x is sampled an unknown $f(x)$ becomes known. As the algorithm terminates exactly when all $f(x)$ are known, and there are only $n = |X|$ unknown x at the start, and as, because it is not resampling, the algorithm only ever samples x for which $f(x)$ is unknown, then the algorithm necessarily terminates after exactly n samples, and we reach a leaf.

We now show that no two paths arrive at the same node. First we note that all paths through the tree start at the root node. Assume we have two paths that at some point separate. If they were to rejoin they must both eventually arrive at the same data set. However, the very fact that they separated means that for some x they produced

different $f(x)$ values. Thus, their data can never be the same, and they will never rejoin at a node. It follows that the graph is a tree.

Finally, as every path necessarily terminates after exactly n steps, and every non-leaf node leads to exactly the same number of immediate children (m), it follows that the tree is balanced.

□

In Figure 1.3 we show how the on-policy behaviour of optimisers can be represented in a tree as shown in the above theorem and proof. The figure shows both a detailed explanatory version of the tree and the equivalent more compact version of the tree in the style we will actually use.

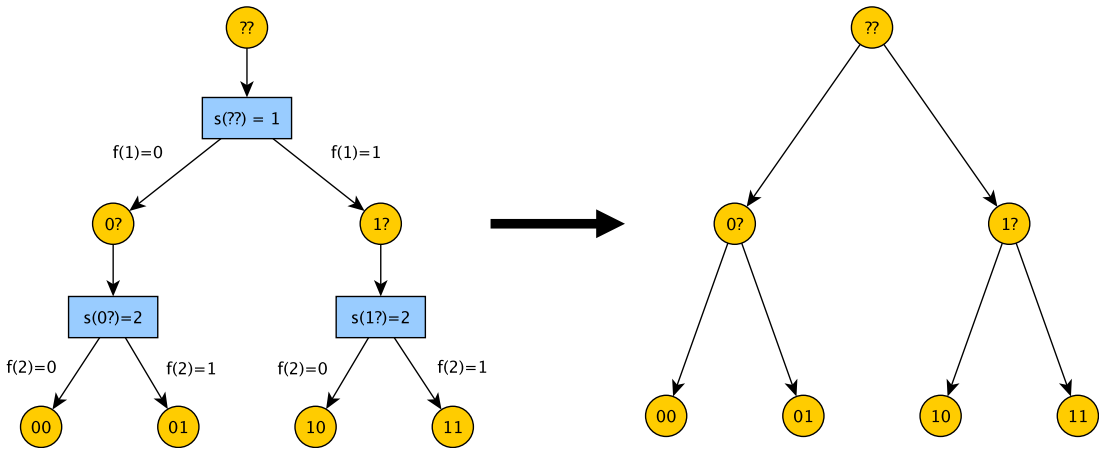

Figure 1.3: Left: The on-policy behaviour of the optimisation algorithm using the sampling policy from Example 5. The yellow circular nodes represent data, the blue rectangular nodes show the sampling policy's decision based on data. Right: the same tree shown more compactly by removing the explanatory details. The tree still contains the same information and is the format we will generally use, for compactness.

Theorem 2 (Tree Representation Details)**.** *The tree representing the on-policy behaviour of an optimiser for functions mapping $X \to Y$ where $|X| = n$ and $|Y| = m$ has $n+1$ layers. If we label these layers $0, 1, \ldots, n$ starting from the root then layer i contains m^i nodes, and the tree contains $1 + m + m^2 + \ldots + m^n$ nodes in total. The final layer consists of m^n leaves with exactly one representing each of the m^n functions $f \in \mathcal{F} = Y^X$.*

Proof. Each node represents particular data $d \in \mathcal{D}$. The root node is always d s.t. $\forall\, x \in X,\ d(x) = ?$. Call this root node layer 0. Each node on the ith layer, for $i \in \{0, \ldots, n\}$,

will represent data with exactly $n-i$ unsampled x values. Thus layer $i=n$ (the $n+1$th layer) will consist of leaves, and will be the final layer.

Every non-leaf node leads to exactly $m=|Y|$ immediate children. We have also seen in the proof of theorem 1 that no node is the child of more that one node. Thus, as in layer zero there is 1 node in layer 1 there will be m, in layer 2 there will be m^2 nodes, and in layer i there will be m^i nodes for i ranging from 0 to n.

It follows from the tree's definition that nodes in layer i have had exactly i X values sampled. And thus the final layer consist of all and only the m^n leaves.

□

In Figure 1.4 we show example on-policy trees for three different domains. It can be seen that even for $|X|=4$ and $|Y|=2$ the tree becomes fairly large. Although trees are possible for any finite X and Y we will generally not be able to show them explicitly!

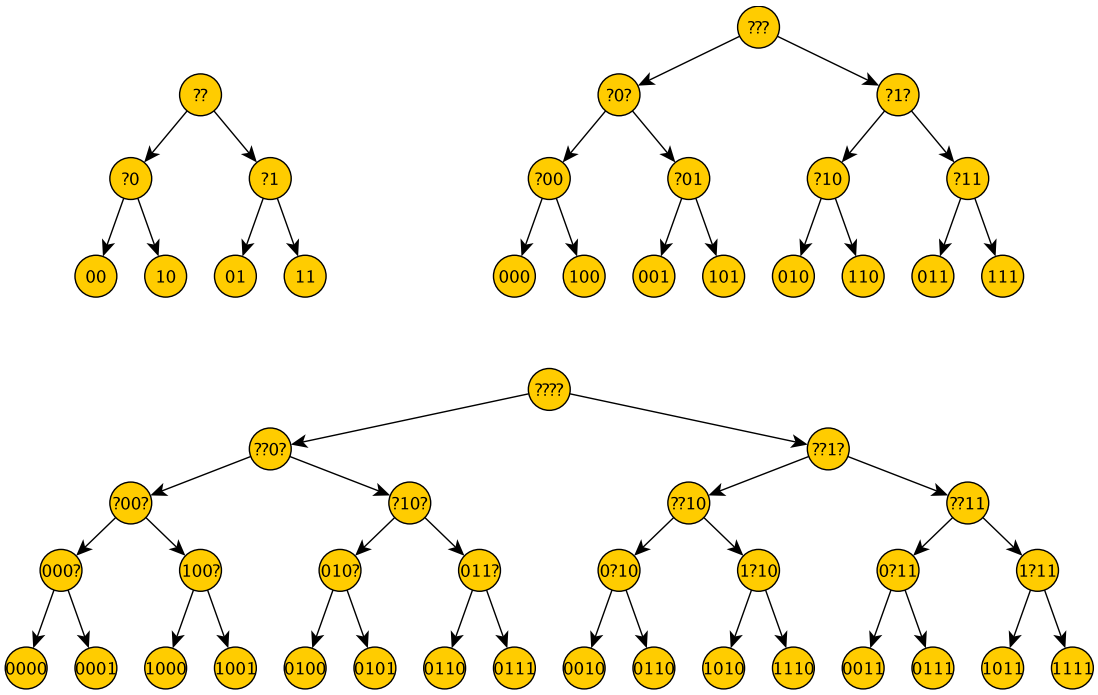

Figure 1.4: Thee example policies, represented as trees. Top left: a policy for functions from $X=\{1,2\}$ to $Y=\{0,1\}$. Top right: a policy for functions from $X=\{1,2,3\}$ to $Y=\{0,1\}$. Bottom: a policy for functions from $X=\{1,2,3,4\}$ to $Y=\{0,1\}$.

We have so far seen that on-policy behaviour can be represented as a tree. We now show that when it comes to the full behaviour it is no longer possible to represent the behaviour as a tree, but we can still represent it as a graph.

Theorem 3 (Full Behaviour Graph Representation)**.** *The full behaviour of a non-resampling optimisation algorithm over functions $f : X \to Y$ where $|X| = n$ and $|Y| = m$ can be represented as a directed graph with $(m+1)^n$ nodes. The graph contains m^n nodes with no outgoing edges, with exactly one representing each of the m^n functions $f \in \mathcal{F}$. All other nodes have exactly m outgoing edges.*

Proof. The full behaviour needs to describe the algorithm's behaviour for every possible data input. Thus, the number of nodes will be exactly the number of possible data sets, which is $(m+1)^n$. As in Theorem 2, each node is either a leaf or has m outgoing edges corresponding to the m possible results of the sample. □

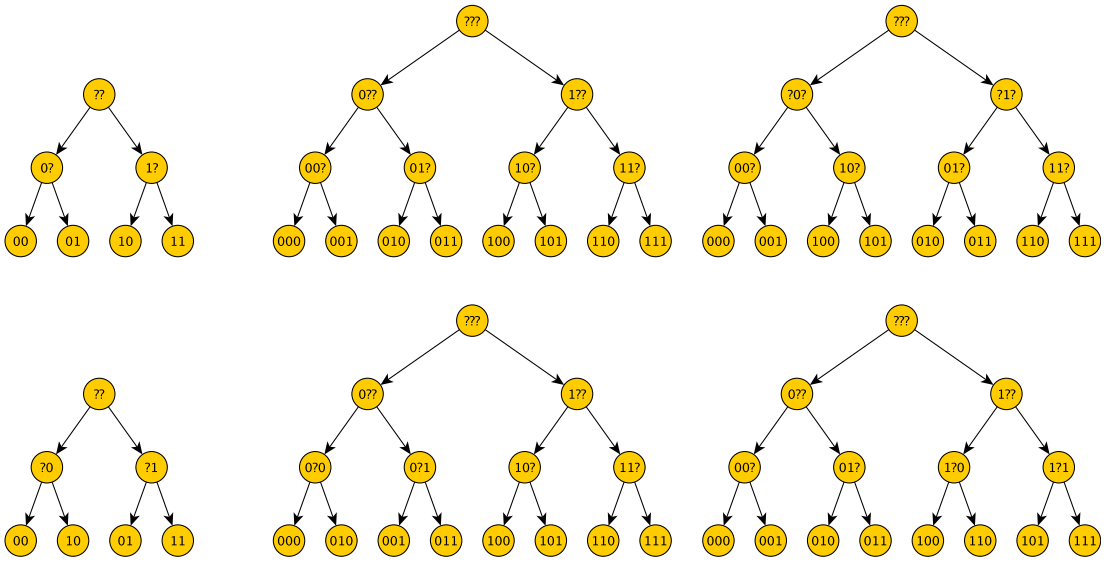

Figure 1.5: Left: Both of the two possible sampling policies on functions $f : \{1,2\} \to \{0,1\}$. Right: Four of the twelve possible sampling policies on functions $f : \{1,2,3\} \to \{0,1\}$.

1.8.5 Enumerating Optimisation Algorithms

We now use counting arguments to calculate the number of sampling policies for full behaviours and for on-policy behaviours. As we will see, the number of unique behaviour grows double exponentially, even if we restrict our attention to only the on-policy behaviour of deterministic non-resampling optimisers.

1.8.5.1 Enumerating On-Policy Behaviours

Here we calculate how many unique on-policy behaviours exist over functions $f : X \to Y$.

Theorem 4 (Number of On-Policy Behaviours)**.** *Let $|X| = n$ and $|Y| = m$. Then there are $\prod_{k=1}^{n} k^{m^{n-k}}$ possible on-policy behaviours for algorithms optimising functions $f : X \to Y$.*

Proof. From above we know that on-policy behaviours can be represented as trees with $n+1$ layers, where the ith layer, for i in $0, \ldots, n$, contains m^i nodes. Nodes in the ith layer have $(n-i)$ unsampled X values to potentially select as the next sample. Thus, layer i has m^i nodes each choosing from $(n-i)$ options, yielding $(n-i)^{m^i}$ possibilities for the layer. Taking the product over all the layers yields $\prod_{i=0}^{n}(n-i)^{m^i}$. Finally, noting that $(n-n)^{m^n} = 1$ and substituting k for $n-i$ we get to the result in the theorem. Thus,

$$\#\text{ of on-policy behaviours} = \prod_{k=1}^{n} k^{m^{n-k}}$$

□

Example 8. *Figure 1.5 shows the $1^{2^1} \times 2^{2^0} = 2$ possible on-policy behaviours when $|X| = 2$ and four of the $1^{2^2} \times 2^{2^1} \times 3^{2^0} = 12$ possible on-policy behaviours for $|X| = 3$. In both cases $Y = \{0, 1\}$.*

1.8.5.2 Enumerating Full Behaviours

We now enumerate the number of distinct full behaviours for optimisation algorithms.

Theorem 5 (Number of Full Behaviours)**.** *Let $|X| = n$ and $|Y| = m$. Then there are $\prod_{k=1}^{n} k^{\binom{n}{k} m^{n-k}}$ possible full behaviours for algorithms optimising functions $f : X \to Y$.*

Proof. First observe that the number of possible data sets containing exactly k unsampled x values is $\binom{n}{k} m^{n-k}$ ($\binom{n}{k}$ for the possible positions of the k unknown $f(x)$ values and m^{n-k} for the possible values of the $n-k$ known $f(x)$).

Presented with such a data set the optimiser must choose to sample one of the k as-of-yet unsampled x values. Thus, the behaviour when k values are unsampled is determined by $\binom{n}{k} m^{n-k}$ separate choices from k options, producing $k^{\binom{n}{k} m^{n-k}}$ possibilities for the layer. Taking the product over all the layers yields the final result:

$$\#\text{ of full behaviours} = \prod_{k=1}^{n} k^{\binom{n}{k} m^{n-k}}$$

□

1.8.5.3 What Fraction of Behaviour is On-Policy?

We have seen that there is on-policy behaviour, and that this is a subset of the full behaviour of an algorithm, in the sense that the a full behaviour describes both the on-policy and off-policy behaviour of an optimiser. There are various distinct full behaviours with the same on-policy behaviour.

If we let $\#OPB$ be the number of on-policy behaviours and $\#FB$ be the number of full behaviours then one immediate question is: what happens to the fraction $\frac{\#OPB}{\#FB}$ as the size of X or Y increase?

The number of nodes describing the on-policy behaviours as a fraction of the number of total nodes in the full behaviour graph behaviours tends to 0 as $|X|$ increases. The growth of $\#OPB$ and $\#FB$ can be seen and compared in Figure 1.6, which makes the above result intuitive.

1.8.6 Paths Down Trees

We have seen that the on-policy behaviour of an optimisation algorithm can be represented as a rooted tree (Theorem 1). We now show that running an optimiser on a particular function corresponds to taking a particular path down the optimiser's tree. Figure 1.7 provides a graphical example of these paths down trees.

Theorem 6 (Paths Down Trees)**.** *When you represent the on-policy behaviour of an optimisation algorithm as a rooted tree, then paths down that tree biject with the functions $f \in \mathcal{F}$, and the path shows the choices that the algorithm will make when optimising the corresponding f.*

Proof. The existence of the bijection follows directly from the fact that the graph structure is a tree, and there is thus only a single path from the route to each leaf. □

1.9 Conclusion

In this chapter we have introduced and defined various key concepts for discussing optimisation. We have also seen that the number of unique optimisation behaviour grows double exponentially, even if we restrict our attention to just on-policy behaviour. Thus considering all possible behaviours explicitly is intractable for all but the smallest toy

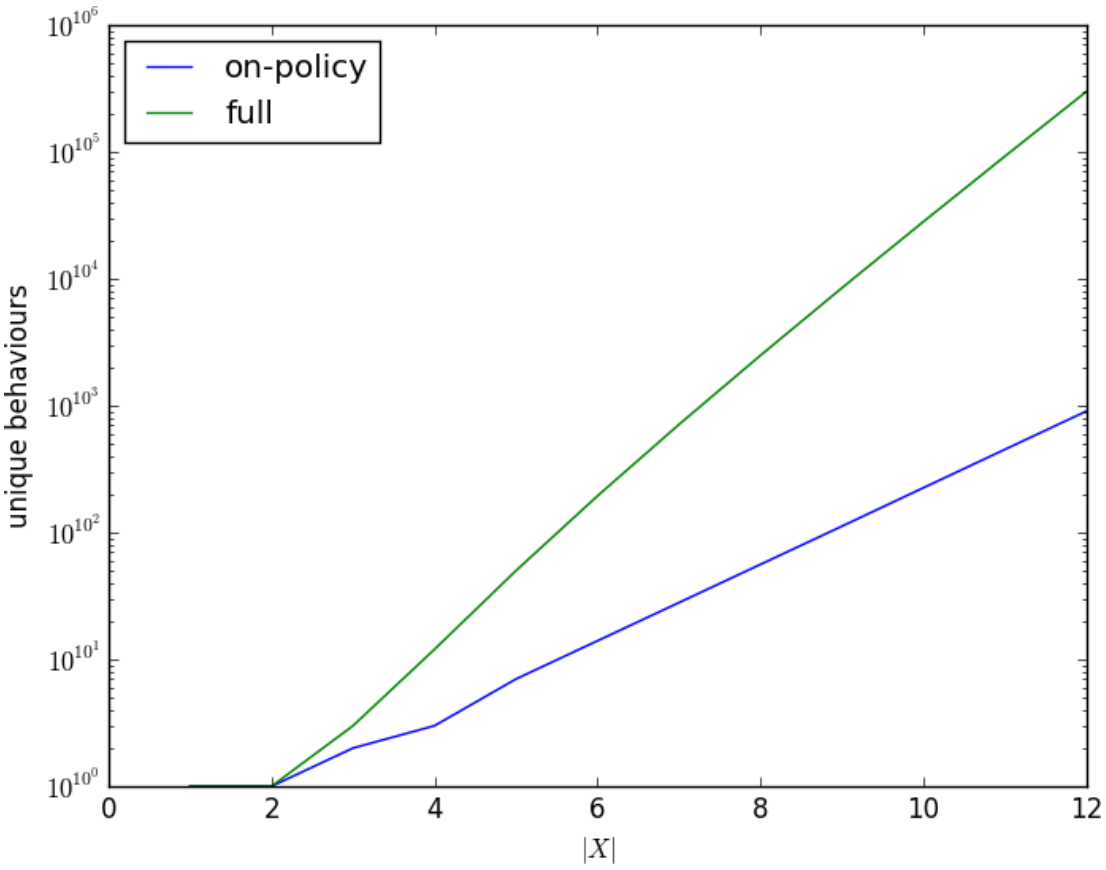

Figure 1.6: The plot shows the doubly exponential growth of the number of optimisation behaviours on binary output (i.e. $Y = \{0, 1\}$) functions as $|X|$ increases. It can be seen that for $|X| = n$ the number of full and on-policy behaviours are approximately $10^{10^{\frac{2n}{5}}}$ and $10^{10^{\frac{n}{4}}}$ respectively.

problems. In fact, describing even a single policy by explicitly writing all input-output pairs as we have been doing up until now is very quickly untenable as the input space grows.

Further, we have shown that any deterministic non-repeating on-policy sampling behaviour can be described by a tree, and that optimisation can be seen as following a path down such a tree. Both of these facts will be used repeatedly in chapters to come.

In the next chapter we start to look at optimisation algorithms in detail. In particular we examine how you might begin to compare and evaluate them.

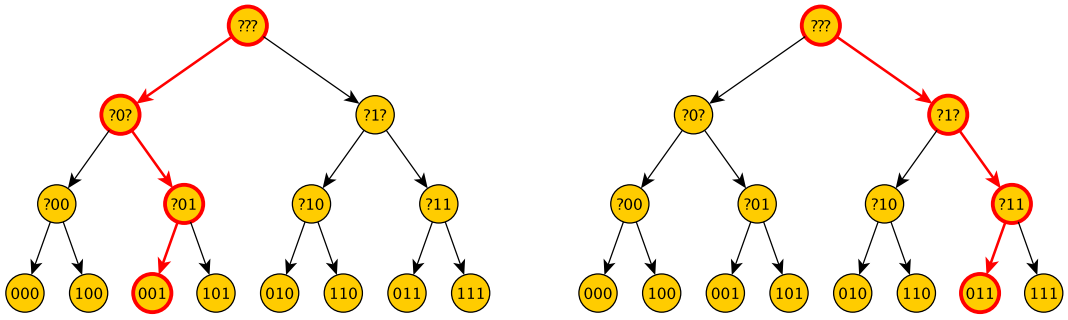

Figure 1.7: Two example paths taken by the optimisation algorithm down its behaviour tree. The left path shows the route when the algorithm is run on $f = 001$ and the right path shows the behaviour on $f = 011$.

Chapter 2

Comparing Optimisers

"...on average hill climbing performs the same as hill descending, even when the goal is function minimization!" — Ho and Pepyne, 2002, [1].

"From a theoretical point of view, comparative evaluation of search algorithms is a dangerous, if not dubious, enterprise." — Whitley, 2014, [2].

2.1 Introduction

In the previous chapter we defined optimisation algorithms and we saw that there are effectively infinite possibilities for optimisation algorithm behaviour. However, so far we have not considered what might make one optimisation algorithm better than another, nor have we discussed what makes an optimisation algorithm *good*. These are the questions we consider in this chapter.

It is not straightforward to say what makers an optimisation algorithm good. It was consideration of this problem of *a priori* optimiser comparison that originally led to the so called no free lunch results we discuss below. Before we discuss the no free lunch results though we first examine two potential methods of optimiser comparison, benchmarking and theoretical assurances. After these short sections, the remainder of the chapter is mainly a review of the no free lunch theorems.

This chapter also includes some new results. Specifically, we consider no free lunch theorems for algorithms with arbitrary stopping conditions in section 2.5.2, we examine the no free lunch theorems in the case of off-policy situations in section 2.6.6, and the case of restricted metrics in 2.6.7. Further we consider the effects of parallelism on optimisation and free lunches in section 2.7.7. Additionally, in section 2.8.2 we discuss how various proofs of optimisation algorithm efficacy can be seen as examples

of restricted setting free lunch results, we provide a new proof regarding algorithm minimax behaviour in section 2.8.7, and we use order statistics to investigate how no free lunch theorems relate to the curse-of-dimensionality in section 2.9.9.

2.2 Benchmarking

Perhaps the most common and straight-forward way to compare optimisation algorithms is though benchmarking. At a high level benchmarking is the comparison of algorithms by comparing their performance on "benchmark" problems.

The details of the specific problems chosen and the definition of performance used are largely case-specific. This said however, performance is usually concerned with some or all of: the speed at which the algorithm finds adequate solutions, quality of solutions found, and how reliably the algorithm finds adequate solutions.

The benchmark problems chosen are generally intended to be varied and representative. A common set of benchmark functions has emerged in the literature, and often algorithm comparison defaults to benchmarking on this common set, or some variation of it. A representative example of such a benchmark set is [3] in which Li et al detail various benchmark problems for use in The Congress on Evolutionary Computation's global optimisation contests.

More recently there has been a trend towards demonstrating the effectiveness of algorithms by using them on real high dimensional non-convex optimisation problems, like deep and recurrent neural network training on various complex data sets. Efficient algorithms for training deep architectures are currently producing cutting edge results in many areas of machine learning, and the problem is thus a very attractive one.

The underlying idea is that, (1) the benchmarks are good representative examples of the kinds of function that will be encountered in real applications, and (2) if optimisation algorithms perform well on the benchmark problems then they are likely to perform well on the real problems of which the benchmarks are representative examples. Both (1) and (2) are potentially false.

2.2.1 Examples

As a simple example of benchmarking we compare the standard particle swarm optimisation algorithm with a heterogeneous particle swarm optimiser over four standard benchmark functions. The results are shown in figure 2.1. We have also included

a benchmarking example from the heuristic optimization literature which is shown in figure 2.2. The figure shows select benchmarking results from [4], in which they aimed to define a standard particle swarm optimization algorithm for comparison purposes.

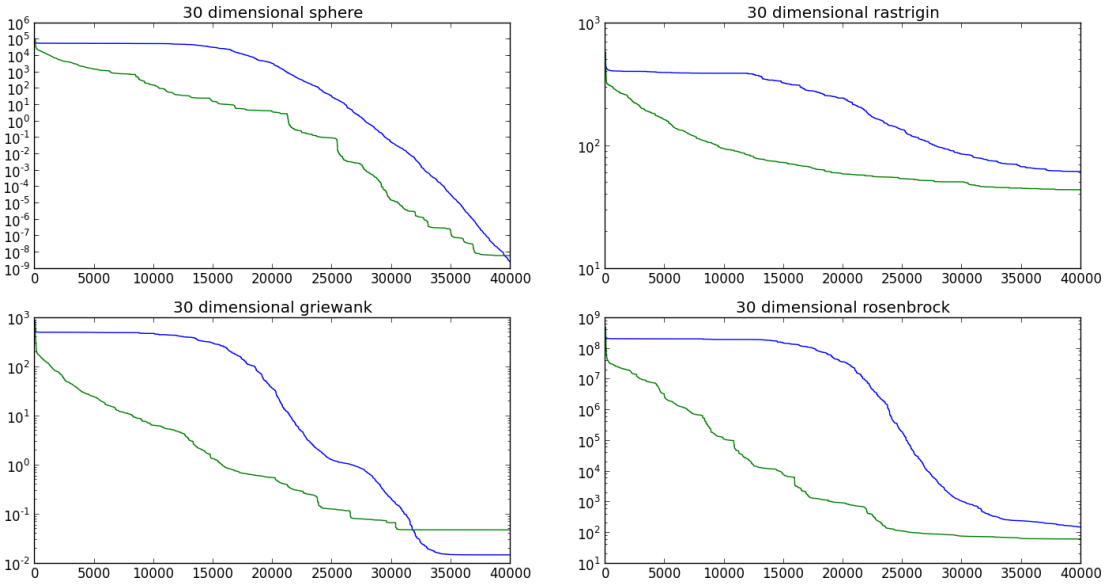

Figure 2.1: Comparison between a standard particle swarm optimisation algorithm (blue) and a heterogeneous particle swarm optimisation algorithm (green) on four common benchmark problems in 30 dimensions. A casual glance would suggest that the heterogeneous particle swarm optimisation algorithm was better, but in fact each optimiser outperform the other in two of the four problem if you consider the final solution found.

2.2.2 Assumptions of Benchmarking

The use of benchmarking to compare algorithms suggests the following underlying assumption: The performance of an optimiser on the test problems is representative of the optimisers performance in some more general sense. When using benchmarking, the way in which the performance is assumed to generalize should ideally be stated explicitly. However, the default claim is roughly that performance generalizes to other problems *of the same kind*. For example, if an algorithm outperforms its competitors on a benchmark set consisting of travelling salesman problems, the hope is that this algorithm is better than its competitors in general at travelling salesman problems. This said, it is worth noting that problem classes can vary widely in breadth, and what makes problem *the same kind* is very much up to interpretation. Often the result of

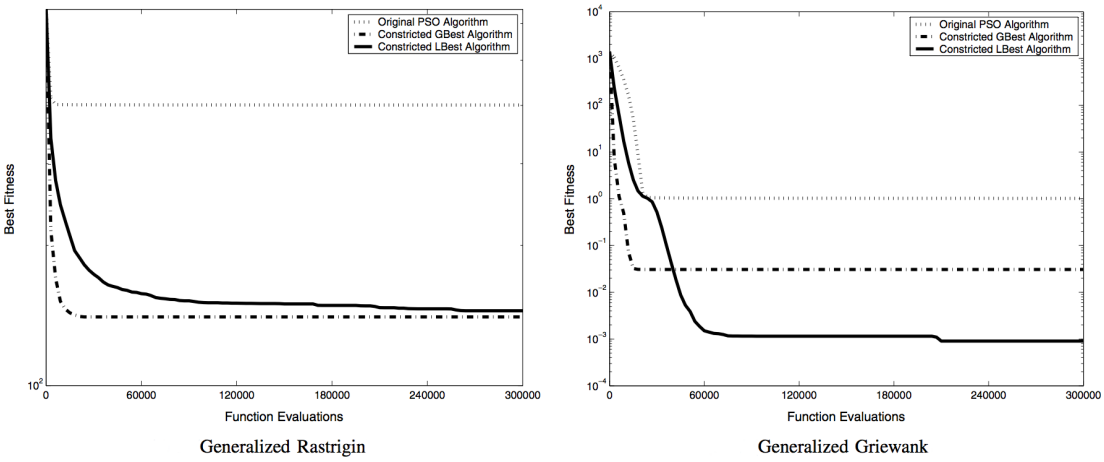

Figure 2.2: Figure taken from [4] showing benchmark results for three varieties of particle swarm optimisation. Like the results in Figure 2.1, these results are also for the 30 dimensional cases.

benchmarking or comparative studies of optimisers is that different algorithms do well on different problems. As we will see in section 2.4, this is all we can really hope for. Benchmarking needs to be done carefully, with the focus shifted to understanding the behaviour rather than ranking it, unless focusing on a particular type of problem.

2.3 Theoretical Assurances

Another way to asses optimisation algorithms is to rigorously prove things about their behaviour. The issue with this approach is that it is technically difficult and requires making assumptions in a more obvious way (you make assumptions when you benchmark, but these are often implicit, whereas here explicit assumptions are generally required). Theoretical assurances are also often concerned with the behaviour in the limit of infinite time. Despite these difficulties and restrictions though, there are various interesting results in this area.

For example, in the Gaussian Process optimisation setting bounds on the cumulative regret (the sum of the differences between the samples and the global optimum) have been proven, provided the function being optimised is sampled from a suitable Gaussian Process [5, 6].

In the next section we will see how these theoretical assurances relate to the no free lunch theorem. In particular, in section 2.8 we examine how often theoretical assurances of algorithms performance are equivalent to proving the existence of a restricted

domain free lunch.

2.4 No Free Lunch Theorems

No free lunch theorems are statements about the non-existence of fully general algorithms. Informally, they show that there is no algorithm that is good at everything. The original no free lunch theorem first appears in Wolpert and Macready's 1995 paper No Free Lunch Theorems for Search [7], but became more widely known through the same authors 1997 paper, No Free Lunch Theorems for Optimization [8]. In this section we present a review of no free lunch results and introduce some extensions. However first we briefly cover the conservation law, which can be seen as a precursor to the no free lunch results.

2.4.1 Conservation Law

A classification problem is a task in which you have a set of objects to classify, and a set of possible classes, and you need to learn how to correctly classify the objects.

A natural question to ask is if there is a best classifier, that is, a classifier that performs better than any other on average. In 1994 Schaffer published a paper to this end in which he described a so-called conservation law for generalisation performance [9]. In the paper Schaffer proves that no classifier's generalisation performance is better than any others over all learning situations. Moreover, he showed that performance of a classifier can be neither created or destroyed, but only differently arranged over potential problems. All classifiers have the same total performance, when aggregated over all possible problems, the only difference between classifiers is their particular distribution of performance over problems. In short he showed that in classification problems aggregate performance is always conserved.

Schaffer's papers slightly pre-dates Wolpert and Macready's famous no-free-lunch papers [7, 8], but builds on earlier work in the area. In [10] Wolpert investigates the potential of stacked classifiers in this classification task, and had already given much attention to the (im)possibility of generalisation without prior knowledge about the problems being considered [11, 12].

The conservation law concerns learning a *classifier*, and although in this chapter we are focusing on optimisation, we mention it here for completeness, and for a rounder picture of no free lunch theorem development. The conservation law can be seen as a

no-free-lunch result for classification. Also this can be seen as a result for a specific optimisation task, a find the optimal classifier problem. Schaffer observed that his results were important for algorithm evaluation:

> *"[E]mpirical success in generalization is always due to problem selection. Although it is tempting to interpret such success as evidence in favour of a learner, it rather shows that the learner has been well applied."*
> - Schaffer, 1994, [9].

Clearly this claim is an important to understand when considering how to compare algorithms. We now move on to Wolpert and Macready's no free lunch theorem. As we will see the conservation law results and interpretations will be echoed, but with application to optimisation.

2.4.2 No Free Lunch

The original No Free Lunch (NFL) theorem for optimization [7, 8] states that no search algorithm can outperform any other under any metric over all problems. There are many formulations of this statement in the literature with different emphasis. Here we present a representative selection, starting with Wolpert and Macready's own characterization:

1. The average performance of any pair of algorithms across all possible problems is identical [8].

2. For all possible metrics, no search algorithm is better than another when its performance is averaged over all possible discrete functions [13].

3. On average, no algorithm is better than random enumeration in locating the global optimum [14].

4. The histogram of values seen, and thus any measure of performance based on it is independent of the algorithm if all functions are considered equally likely [15].

5. No algorithm performs better than any other when their performance is averaged over all possible problems of a particular type [16].

6. With no prior knowledge about the function $f : X \to Y$, in a situation where any functional form is uniformly admissible, the information provided by the value

of the function in some points in the domain will not say anything about the value of the function in other regions of its domain [17].

As this selection shows, NFL allows a broad range of assertions. We will now formally state and prove the NFL theorem, and then show how each of the above characterisations naturally result. We start with two preliminary definitions:

Definition 11 (Traces)**.** *The trace of a search algorithm A running on a function f is the ordered list of (x,y) pairs sampled (where $y = f(x)$). We write $T_A(f)$ for the trace of algorithm A running on function f. We also define $T_A^m(f)$ to be the trace after m function evaluations. It follows that if n is the size of the domain of f then $T_A^n(f) = T_A(f)$. We call $T_A(f)$ the (full) trace and $T_A^m(f)$ a partial trace for any $m < n$. Let $\mathcal{T}$ be the set of all the traces that algorithm A produces on functions mapping X (where $|X| = n$) to Y, that is:*

$$\mathcal{T}_A = \cup_{f \in \mathcal{F}} \cup_{m=0}^{n} T_A^m(f)$$

We also define a trace of just the inputs $T_A(f)_x$ and a trace of just the outputs $T_A(f)_y$ as ordered lists of just the x and y values respectively from the full trace. We call these the input trace and the output trace. The output trace is sometimes called the performance vector or range trace in the literature. Similarly, we define $\mathcal{T}_{Ax}$ and $\mathcal{T}_{Ay}$ all possible input traces and all possible output traces respectively.

Example 9 (Traces)**.** *Let $X = \{1,2,3\}$ and $Y = \{0,1\}$ and let A be an optimisation algorithm with $A(???) = 2$, $A(?0?) = 3$ and $A(?1?) = 3$ (A is shown graphically in Figure 2.3). Let $f : X \to Y$ with $f = 010$. Then $T_A(f) = \{(2,1),(3,0),(1,0)\}$, $T_A^2(f) = \{(2,1),(3,0)\}$, $T_A(f)_x = \{2,3,1\}$ and $T_A(f)_y = \{1,0,0\}$, $\mathcal{T}_{Ay} = \{\{0,0,0\}, \{0,0,1\}, \{0,1,0\}, \{0,1,1\}, \{1,0,0\}, \{1,0,1\}, \{1,1,0\}, \{1,1,1\}, \{0,0\}, \{0,1\}, \{1,0\}, \{1,1\}, \{0\}, \{1\}, \{\}\}$.*

As noted in [18] search algorithms have no "intrinsic purpose" and their behaviour needs to be qualified by some external metric. A metric provides a way to evaluate an algorithms performance. We now make this idea of a metric precise.

Definition 12 (Optimisation Metric)**.** *An optimisation metric M is a function that maps output traces to $\mathbb{R}$. A metric can be thought of as assigning a value, or score, to an output trace, $M : \mathcal{T}_y \to \mathbb{R}$.*

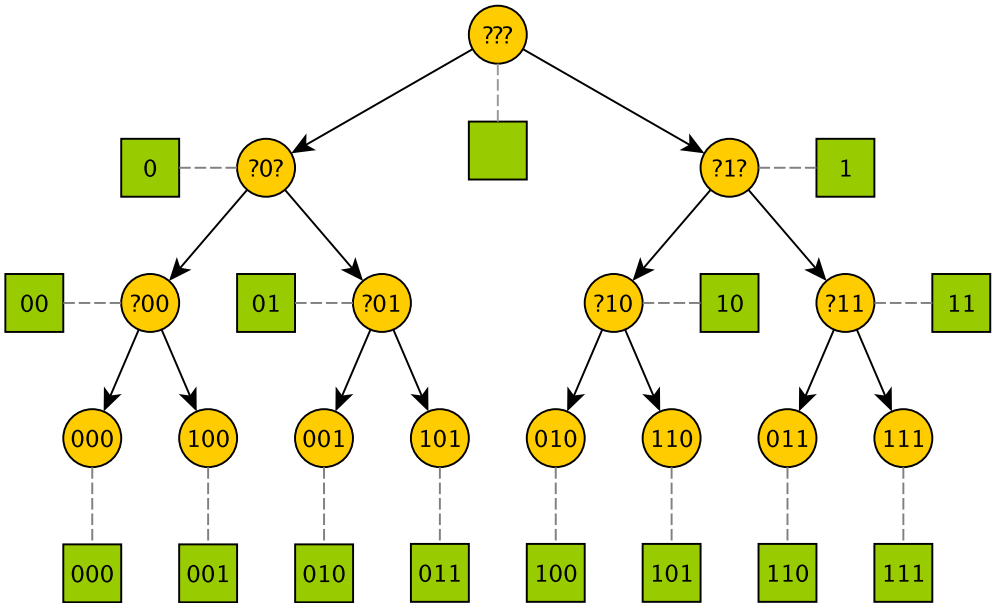

Figure 2.3: The yellow nodes show the behaviour tree for a search algorithm on functions between $\{1,2,3\}$ and $\{0,1\}$. The green squares attached to each yellow node show the corresponding trace at that point. If we let A be the search algorithm represented by the tree, then it can see that, for example, $T_A(010)_y = 100$ by reading the trace at the 010 leaf.

An alternative but equivalent way to think of a metric is as a map from an algorithm and a function to a rating of how well the algorithm performs on that function, that is, $M^* : \mathcal{A} \times \mathcal{F} \to \mathbb{R}$ *where* $M^*(A, f) = M(T_A(f)_y)$.

Example 10 (Optimisation Metrics)**.** *A simple metric for measuring minimisation performance is a function that returns the minimum Y value in the trace (i.e. the smallest y value sampled so far). In fact, although simple, this metric is often used to compare optimisers, especially in cases where the true global minimum of the function is unknown.*

We are now in a position to formally state and prove NFL. We follow the style of proof used by English [19, 18, 20], as this is in our opinion clearest, and naturally yields various generalisations to the theorem. It involves explicit reference to the representation of optimisers as trees. Another particularly clear approach to the proof is using the "fundamental matrix" method from [1], which is equivalent to our approach here. We have chosen a tree based method because, in our opinion, it makes the sequential decision aspect of the problem more explicit. However, we strongly recommend [1] to any readers wanting an alternative approach.

Theorem 7 (NFL)**.** *All search algorithms produce the same set of traces when run over all possible functions between two finite sets. More formally, let* $Y^X = \{f \mid f : X \to Y\}$*, where* X *and* Y *are arbitrary finite sets. For all search algorithms* A, B*,* $\{T_A(f)_y \mid f \in Y^X\} = \{T_B(f)_y \mid f \in Y^X\}$.

Proof. As we have shown in Section 1.8.4, the behaviour of a search algorithm is uniquely representable as a tree. This tree has $|Y|^{|X|}$ leaves and the trace at each leaf is unique. However, by a counting argument there are only $|Y|^{|X|}$ possible distinct traces of length $|X| = n$. Therefore, every search algorithm produces the same set of traces, namely every possible trace exactly once. □

It is worth noting that the proof above is very succinct, this is in part due to the reference to the proof that the behaviour of a search algorithm is uniquely representable as a tree, but also partly a result of the decision tree proof style. We now return to the various formulations of the no free lunch result found in the literature, and show that they all follow from the proof above.

1. The average performance of any pair of algorithms across all possible problems is identical [8].

   → From our no free lunch proof we know that any pair of algorithms produce the same set of traces over all functions. The set of all traces being equal implies that the multi-sets of performance values are equal (regardless of metric), which in turn implies that the mean values are equal.

2. For all possible metrics, no search algorithm is better than another when its performance is averaged over all possible discrete functions [13].

   → This is equivalent to statement one, but with explicit reference to the independence of the result on the metric chosen. The reference to discrete functions is emphasising that X and Y are finite.

3. On average, no algorithm is better than random enumeration in locating the global optimum [14].

   → This claim has two interpretations, both of which are in fact true. Firstly we could assume that "random enumeration" means a randomly chosen, but thereafter fixed, order of X values which we follow when exploring a function. This order of inspection is just some particular deterministic optimisation algorithm, and so the claim follows from NFL. The second possible interpretation of

"random enumeration" is that we randomly choose where to sample whenever we need to make a choice. This leads to a stochastic algorithm, and as we will show in Theorem 11, a no free lunch theorem holds in this case too.

4. The histogram of values seen, and thus any measure of performance based on it is independent of the algorithm if all functions are considered equally likely [15].

   → The set of all traces being equal implies that the multi-sets of values are equal (regardless of metric). It is this multi-set which is meant by the histogram (they are both just sets where an element can occur more than once, i.e. elements have a count or frequency). Also, "All functions being considered equally likely" is an phrasing which points towards a probabilistic NFL representation (see subsection 2.7.2).

5. No algorithm performs better than any other when their performance is averaged over all possible problems of a particular type [16].

   → Here "problems of a particular type" means functions mapping between some input set X and some output set Y. Assuming that X and Y are finite then this is exactly the set of problems considered in the above proof.

6. With no prior knowledge about the function $f : X \to Y$, in a situation where any functional form is uniformly admissible, the information provided by the value of the function in some points in the domain will not say anything about the value of the function in other regions of its domain [17].

   → This formulation starts to use a probabilistic framework. More specifically it invokes (the lack of) a Bayesian prior as a way of understanding the no free lunch result. This Bayesian interpretation is, we believe, the right one, in the sense of being clear and informative.

2.5 Basic No Free Lunch Extensions

Having stated and proven the original no free lunch theorem, we now present some additional results and observations that are of interest in themselves, and will also be used in refinements of NFL below.

Theorem 8 shows that every algorithm is equivalent when full traces are considered. We first generalise to the case where we stop the optimisation after m steps, then we

generalise further to the case of arbitrary stopping conditions. After this we give a proof that the NFL result still holds if we allow stochastic sampling policies.

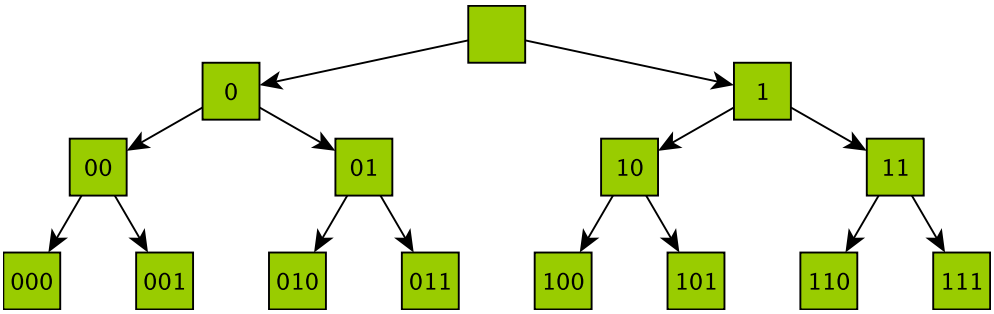

Figure 2.4: The behaviour tree showing only the trace values. When considering just the traces, all algorithms have the same behaviour tree (see Theorem 9). What differentiates algorithms is that for different algorithms a given function will produce different *paths* in this tree.

2.5.1 m-Step No Free Lunch

In real applications, optimisation algorithms are not usually run until the entire domain has been sampled. In the orignal no free lunch papers by Wolpert et al [7, 8] they show that the no free lunch theorem still applies if algorithms are run for some fixed number of steps. We restate this theorem here and prove it using tree representations.

Theorem 8 (m-step NFL)**.** *All search algorithms produce the same set of traces when run over all possible function between two finite sets for m steps. More formally, let $Y^X = \{f \mid f : X \to Y\}$, where X and Y are arbitrary finite sets. For any search algorithms A, B, $\{T_A^m(f)_y \mid f \in Y^X\} = \{T_B^m(f)_y \mid f \in Y^X\}$. In fact they produce the same multi-set, in that every possible trace appears the same number of times, with the exact value depending on m and $|X|$.*

Proof. If we prune the full behaviour tree after m steps the resulting tree will have Y^m leaves. Each leaf has a unique trace and the traces are of length m. Thus, there are only Y^m possible partial traces, and so each partial trace must be present exactly once in the leaves. This is the case for all search algorithms.

It remains to show that the multi sets are the same. First observe that each partial trace appears in the multi set once for each leaf it would have lead to in the full tree. Next note that each leaf in the m-step truncated tree would lead to the same number

of leaves in the full tree (as every non-leaf node in the full tree branches to the same number of nodes in the next layer). Thus, all traces appear the same number of times in the multi-set. Moreover, if $|X| = n$ and we truncate the tree after m steps, then each trace will occur 2^{n-m} times in the multi-set. In particular, if $m = n$ then each trace appears exactly once as expected. □

We defined the behaviour tree for an optimiser in Section 1.8.4. We now define a similar but distinct tree representation, the trace tree. Essentially the behaviour tree show both the x and y values of the algorithms samples, the trace tree in contrast only shows the y values.

Definition 13 (Trace Tree)**.** *Let A be an optimisation algorithm for functions $f : X \to Y$. The trace tree for A is the behaviour tree with the nodes labelled with the trace, rather than the data.*

An example trace tree is given in Figure 2.4. It is the trace tree of the optimisation algorithm in Example 2.3, the behaviour tree of the same algorithm is shown in Figure 2.3.

We now show that trace tree only depends on the range and the size of the domain of the functions being optimised. The detail that we lose when we switch from behaviour trees to trace trees is exactly the detailed that differentiated the optimisers.

Theorem 9 (Identical Trace Trees)**.** *All optimisation algorithms for functions mapping X to Y produce the same trace tree. In fact, all optimisation algorithms for functions mapping Z to Y also produce the same trace tree, as long as $|Z| = |X|$.*

Proof. Consider an optimisation algorithm A for functions $f : X \to Y$. We will show that A's trace tree does not depend on any of the specific details of A, and thus would be the same for any optimisation algorithm for functions $f : X \to Y$.

From the definition of a trace tree we know that the algorithm's trace tree has the same structure as the algorithm's behaviour tree. In particular as we are only considering non-resampling optimisers we know that the trace tree will have $|X|+1$ layers, and that each node in the tree will be either a leaf or the parent of exactly $|Y|$ other nodes.

Now we simply observe that if we consider a non-leaf node in the trace tree then we can detail the node's children without knowledge of A. Suppose the node we are considering has trace $y_1, y_2, \ldots, y_m$, then its $|Y|$ children will have the traces $y_1, y_2, \ldots, y_m, y$ for each $y \in Y$.

Next we note that the only influence the domain X has on the trace tree is in setting the number of layers to be $|X|+1$, thus it is only the size of the domain that is important and the trace tree will be the same for any domain of that size. □

Corollary 1. *The leaves of the trace tree are all possible problem functions.*

This was shown in Theorem 2 but is restated as a corollary above, as it is important in the proofs to follow.

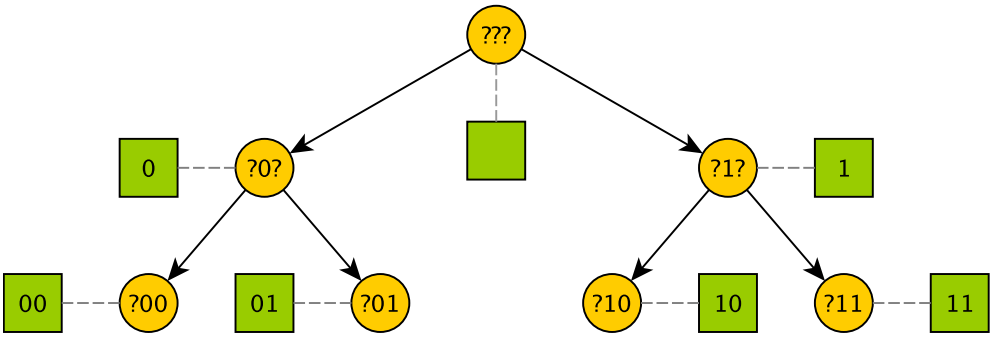

Figure 2.5: The tree in figure 2.4 after $m = 2$ steps. As $|Y| = |\{0,1\}| = 2$ there are $|Y|^m = 2^2 = 4$ leaves, each having one of the four possible two bit traces.

Continuing our generalisations of NFL to early stopping scenarios, we now consider the more general situation in which the search process can terminate based on the results so far, rather than simply after a fixed number of steps. As we will see, a no free lunch result still pertains.

2.5.2 Stopping Condition No Free Lunch

A stopping condition is a rule for when to stop the optimiser. We make the concept of stopping condition formal and then we show that the set of traces over all functions is algorithm independent for any stopping condition. In other words, a NFL result still holds.

Definition 14 (stopping condition)**.** *A stopping condition is a function mapping traces to either* 0 *or* 1*,* $S : \mathcal{T} \to \{0,1\}$*. The stopping condition can be thought of as looking at the results of the optimisation algorithm so far and deciding whether to continue searching. After each function evaluation an optimisation algorithm using a stopping condition* S *evaluates* S *on its current trace* T *and then stop iff* $S(T) = 1$*.*

Using the above definition we now state and prove a no free lunch result for arbitrary stopping conditions.

Theorem 10 (stopping-condition NFL)**.** *For any stopping condition S, all search algorithms produce the same set of traces when run over all possible functions between two finite sets. More formally, let $Y^X = \{f \mid f : X \to Y\}$, where X and Y are arbitrary finite sets. For any search algorithms A, B, $\{T_{A|S}(f)_y \mid f \in Y^X\} = \{T_{B|S}(f)_y \mid f \in Y^X\}$, where $T_{A|S}(f)_y$ is the output value trace generated by algorithm A using stopping condition S running on function f.*

Proof. From Theorem 1, we know that the behaviour of an algorithm can be represented by a tree. A stopping condition can be seen as a pruning of this tree. Whereas in the m-step NFL proof we cut each branch after the same number of steps, a stopping condition can potentially prune branches after differing numbers of steps.

As we have seen above in Theorem 9, all algorithms produce the same trace tree (see Figure 2.4). The stopping condition leads to a pruning of this trace tree, specifically we prune all the children from any leaf with a trace T such that $S(T) = 1$.

A particular stopping condition, then, leads to a particular pruning. As the trace tree and the pruning are both algorithm independent the resulting pruned trace tree will be the same for all algorithms, and thus its leaves (the final traces produced) will be the same. It follows that, for any search algorithms A, B, $\{T_{A|S}(f)_y \mid f \in Y^X\} = \{T_{B|S}(f)_y \mid f \in Y^X\}$. □

2.5.3 Stochastic No Free Lunch

We now state and prove the basic no free lunch result for stochastic optimisation algorithms. The stochastic case was considered in Wolpert and Macready's original papers [7, 8], although the proof provided here is a succinct one.

Definition 15 (Stochastic Optimiser)**.** *A stochastic optimisation algorithm is an optimiser that uses a stochastic sampling policy to choose where it samples. Previously a sampling policy was a function mapping $s : \mathcal{D} \to X$. In the case of a stochastic optimiser the sampling policy is instead a function $s : \mathcal{D} \to P_X$ where P_X is a probability distribution over X. To select the next x to sample given data d you draw a sample from the distribution $s(x)$.*

Theorem 11 (stochastic NFL)**.** *Let $Y^X = \{f \mid f : X \to Y\}$, where X and Y are arbitrary finite sets. For any stochastic search algorithms A, B, if we sample f uniform randomly from Y^X then $P(T_A(f) = t) = P(T_B(f) = t) = \frac{1}{|Y|^{|X|}}$ for all full length traces $t \in \mathcal{T}$.*

Proof. Let A be a stochastic optimisation algorithm. This stochastic behaviour is equivalent to sampling a deterministic algorithm from some probability distribution over algorithms, then running that. However, we know from the original no free lunch result that regardless of the deterministic algorithm chosen, each trace is equally probable when each function is equally probable, thus each traces has probability $\frac{1}{|Y|^{|X|}}$ regardless of the stochastic optimiser used. □

2.6 Refined and Generalised No Free Lunches

Since their original publication the NFL theorems have been augmented and specialised in various non-trivial ways. In this section we survey these extensions, providing intuitive explanations of the results, as well as proofs and examples. First though we cover some preliminary results, showing that search algorithms can be understood as bijections between functions, and that the no free lunch results are representation invariant.

2.6.1 Optimisation Algorithms are Bijections

In this section we show that an optimisation algorithm can be seen as defining a function mapping Y^X to itself, and that this function is a bijection, and thus a permutation. We also look at the behaviour of an optimiser when run on a particular function f, and we will see that the output trace produced can be interpreted as a function permutation of the input f. These sorts of results, relating optimisation algorithms to permutations, are worked through in detail in [21]. We start with a definition.

Definition 16 (Shuffle Permutation)**.** *$\pi : Y^X \to Y^X$ is a shuffle permutation if it is a bijection and, if $\pi(f) = g$ then there exists a permutation $\phi : X \to X$ such that $\forall x \in X,\ f(x) = g(\phi(x))$. That is, g is a function permutation (see Definition 3) of f. Intuitively, a shuffle permutation maps a function to new function with the same output values, but rearranged to correspond to different inputs.*

Now we will show the sense in which an optimisation algorithm can be thought of as a map from Y^X to itself. Recall that without loss of generality we assume that

$X = \{1, 2, \ldots, n\}$. Recall also that $T_{A_y}(f)$ is the full output trace of optimiser A running on function f. Thus $T_A(f)_y$ is an ordered list of n output (i.e. $y \in Y$) values. Given this trace a new function, g, can be defined as $g(i) = T_{A_y}(i)$, where $T_{A_y}(i)$ is just the y value of the i-th position in the trace.

We now show that any optimisation algorithm naturally implies a map from Y^X to itself, and in fact this map is a shuffle permutation.

Theorem 12 (Optimisation Algorithms Imply Shuffle Permutations)**.** *Let A be an arbitrary search algorithm, then T_{A_y} is a bijection $T_{A_y} : Y^X \to Y^X$. Further this bijections is a shuffle permutation.*

Proof. That the implied map is a bijection follows from Theorem 1. Next we note that from their definition the optimisers are non-resampling and eventually sample every point. It follows that for any input function the output trace is just a reordering of the function's y values. Thus, the bijection is a shuffle permutation. □

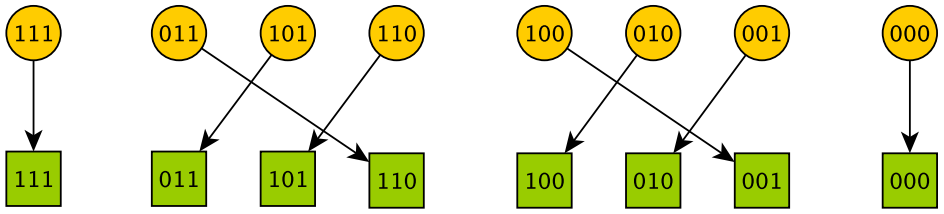

Figure 2.6: The bijection induced by the search algorithm shown in Figure 2.4. As the functions and the traces are both the same set (i.e. length three binary strings) the bijection is in fact a permutation. Also, as this permutation is a shuffle permutation the number of 1s and 0s in each input is preserved in the output.

2.6.2 Representation Invariance

The way a problem is represented is often considered when trying to find optimal solutions. However, an interesting corollary of the no fee lunch theorem is that the representation doesn't matter when considering the ensemble of all possible problems. In other words, there is a representational no free lunch: no representation scheme is better than any other under any metric for any search algorithm when average performance over all problems is considered. This representation invariance was made explicit in [22].

Theorem 13 (Representation Invariance [22]). *Given a function $h : X \to Y$ we can re-represent the problem by introducing a set C and a surjective map $g : C \to X$ and then considering a new function $f : C \to Y$ where $f(c) = h(g(c))$. As C is surjective then we know $|C| \geq |X|$. Then for any optimisers A, B,*

$$|C| = |X| \implies \{T_A(h)_y | h \in Y^X\} = \{T_B(h)_y | h \in Y^X\} = \{T_A(f)_y | f \in Y^C\}$$

$$|C| > |X| \implies \{T_A(f)_y | f \in Y^C\} = \{T_B(f)_y | f \in Y^C\}$$

A full proof can be found in [22], however it can be seen that the case when $|C| = |X|$ follows directly from Theorem 9.

2.6.3 Sharpened No Free Lunch

The Sharpened No Free Lunch Theorem (SNFL) was first presented in [23]. Whereas the original no free lunch theorem is a statement about algorithms on the set of all functions, SNFL shows that the result still holds even when we restrict consideration to certain subsets of function. In particular, SNFL states that all optimization algorithms are equivalent over any subset of functions closed under permutation (see Definition 4 in Chapter 1).

Theorem 14 (SNFL [23]). *Let $G \subset Y^X$ be closed under permutation, then for any search algorithms A, B, $\{T_A(f)_y \mid f \in G\} = \{T_B(f)_y \mid f \in G\}$.*

Proof. We have seen in Theorem 12 that the output trace produced when an optimiser is run on a function f is always some permutation of f. In the same theorem we also saw that for any optimisation algorithm A, for any two functions $f, g \in \mathcal{F}$ $f \neq g \implies T_A(f) \neq T_A(g)$. From these two facts it follows that if the input set is permutation closed, then, regardless of the optimisation algorithm used, the set of output traces will be this same set of functions. In other words, any optimisation algorithm is a permutation on any permutation closed set of functions. $\{T_A(f) \mid f \in G\} = G$ for any optimisation algorithm A. It follows that $\{T_A(f) \mid f \in G\} = \{T_B(f) \mid f \in G\}$. □

Many researchers have examined the realism of the closed under permutation condition for real problems. In particular Igel and Toussaint in [24] show that the proportion of subsets of functions that are closed under permutation tends to zero double-exponentially fast as the domain of the functions increases. A more general consideration of the realism of no free lunch assumptions is given in Section 2.9.1.

2.6.4 Focused No Free Lunch

The Focused No Free Lunch Theorem (FNFL) is an extension of SNFL. Essentially it shows that, when only considering a restricted set of algorithms, a (potentially very small) subset of the permutation closure of a test function is enough for NFL to hold. This result was first presented in [13]. Intuitively, because of the restriction to a subset of algorithms a more focused result is possible as there are fewer requirements to satisfy.

Theorem 15 (FNFL [13])**.** *Let* β *be a set of test functions,* $\beta = \{f_1, f_2, \ldots, f_m\}$*, and let* $\mathcal{A}$ *be a set of optimisation algorithms,* $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$*. Then there exists a "focused set"* $F_{\mathcal{A}}(\beta)$*, with* $\beta \subseteq F_{\mathcal{A}}(\beta) \subseteq \beta_{CUP}$ *such that all algorithms in* $\mathcal{A}$ *have the same average performance over* $F_{\mathcal{A}}(\beta)$*. Moreover, this focused set* $F_{\mathcal{A}}$ *can potentially be much smaller than* β_{CUP} *(note* β_{CUP} *is just the permutation closure of* β*).*

The proof can be found in [13], here we omit the proof and instead include some simple illustrative examples (these examples serve as proofs for the specific $\mathcal{A}$ and β discussed, the paper proves the theorem for arbitrary $\mathcal{A}$ and β).

Example 11 (Simple FNFL Example)**.** *Consider optimising functions mapping between* $\{1,2,3,4,5\}$ *and* $\{0,1\}$*. Let* $\beta = \{01010\}$*, call this function* f_1 *(i.e.* $f_1 = 01010$*). Consider two optimisation algorithms, A, which inspects the function from left to right, and B, which inspects the function from right to left. Clearly* $T_{A_y}(f_1) = 01010 = T_{B_y}(f_1)$*, and thus* $F_{\mathcal{A}}(\beta)$ *for* $\mathcal{A} = \{A, B\}$ *is just* $\{01010\}$*.*

Example 12 (Second FNFL Example)**.** *Again consider optimising functions mapping between* $\{1,2,3,4,5\}$ *and* $\{0,1\}$*. Let* $\beta = \{00110\}$*, let* $f_1 = 00110$*. Consider two optimisation algorithms, A, which inspects the function from left to right, and B, which inspects the function from right to left. Clearly* $T_{A_y}(f_1) = \{00110, 01100\} = T_{B_y}(f_1)$*, and thus* $F_{\mathcal{A}}(\beta)$ *for* $\mathcal{A} = \{A, B\}$ *is* $\{00110, 01100\}$*.*

2.6.5 Almost No Free Lunch

Another important extension to the no free lunch theorem is the so called Almost No Free Lunch Theorem. The Almost No Free Lunch Theorem (ANFL) shows that if an optimiser performs well on a given function then there is a function of similar complexity on which it performs badly [25].

Theorem 16 (ANFL [25])**.** *Let H be a randomized search strategy, $X = \{1,\ldots,2^n\}$, $Y = \{1,\ldots,N\}$ and $f : X \to Y$. Define $c = 2^{n/3}$. Then there exist at least N^{c-1} functions $g : X \to Y$ that differs from f on at most c inputs, such that the probability that H finds the optimum of g within c steps is less than or equal to c^{-1}.*

A proof is given in [25]. Functions of similar complexity here means any of evaluation time, circuit size representation, and Kolmogorov complexity.

2.6.5.1 Example

Let $n = 6$ and $N = 2$. Then $X = \{1,2,3,\ldots,64\}$ and $Y = \{0,1\}$. In this case ANFL asserts that there are at least $2^3 = 8$ functions $g : X \to Y$ that agree with f on all but at most 4 inputs such that H finds the optimum of g within 4 steps with probability bounded above by $\frac{1}{4}$.

2.6.6 Off-Policy No Free Lunch

Here we introduce the Off-Policy No Free Lunch Theorem (OPNFL). As its name suggests OPNFL is an extension of the no free lunch theorem to "off-policy" behaviour. Previous no free lunch research has generally focused on the behaviour of algorithms *starting with no data*. OPNFL extends consideration to the case where algorithms potentially start with some data about the function being optimised. As we will see, the potential inclusion of off-policy decisions means that the optimisation algorithms can no longer be represented as trees, but a no free lunch result still holds.

An example representation of full-policy behaviour as a graph is given in figure 2.7. In the figure the off-policy nodes are shown, which aren't accessible in the normal running of the optimiser, OPNFL is concerned with cases where the optimisation process potentially begins at on of these off-policy nodes.

Theorem 17 (OPNFL)**.** *Running an optimisation algorithm on a uniform randomly selected function starting with a uniform randomly selected compatible partial trace results in every full trace being produced with equally probability, regardless of the optimisation algorithm used.*

Proof. First note that if you uniform randomly select a function and start with no data (i.e. just run the optimiser normally) then regardless of the optimisation algorithm used, every trace is equally likely. This is just a reformulating the original no free lunch result an so follows directly from Theorem 8. Thus, we need to show that if we

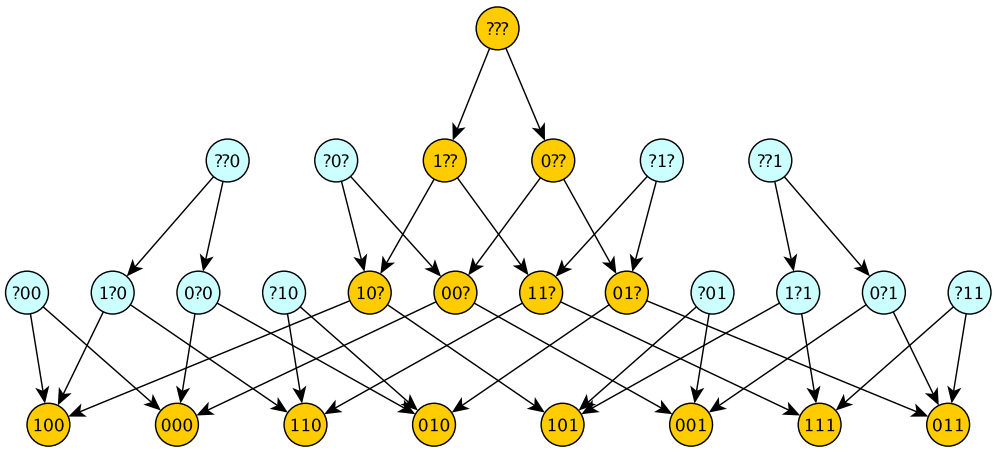

Figure 2.7: An example full behaviour. Off-policy decisions are shown as light blue nodes and on-policy decisions are shown as yellow nodes. Note that full behaviour is representable as a directed graph (as in this diagram), but not as a tree. This is because paths from off-policy and on-policy nodes converge.

uniform randomly select any function compatible partial trace and start optimisation with that data, this doesn't change the expectation over traces. By definition the partial trace we start with is uniformly selected, so we just need to show that the rest of the trace is uniformly distributed over possibilities. However, this is a direct result of the set of optimisers being exactly the set of all permutations.

□

2.6.7 Restricted Metric No Free Lunch

Towards the end of [18] English notes the need for a no free lunch theory for the case of restricted metrics. There has been some work towards this goal, for example in [26] they restrict attention to maximization and shows that the correspondence between no free lunch holding and the set of functions considered being closed under permutation breaks down when you only consider optimisers running for m-steps. Spefically they show that under a maximization metric, the set of functions being closed under permutation is not necessary for no free lunch type results. This is an example of a restricted metric free lunch in an m-step optimisation setting.

Here we provide more work on restricted metric free lunches. In this section we introduce a restricted metric no free lunch theorem (RNFL) as an extension of the FNFL. The original NFL theorem and its successors consider arbitrary performance metrics, or equivalently all performance metrics. Here we show how a restriction on

the set of metrics considered — a choice of a single metric for instance — yields NFL results on subsets of functions. This is similar to FNFL, except as well as considering restricted sets of algorithms and functions it also considers a restricted set of metrics. The key idea is that we compare the multi-sets of scores assigned to traces rather than the multi-sets of traces themselves. The sets of scores often has fewer unique elements (and they can never have more) and thus there are more situations in which no free lunch results hold. Figure 2.8 shows how metrics can reduce the set of traces to a smaller set of scores.

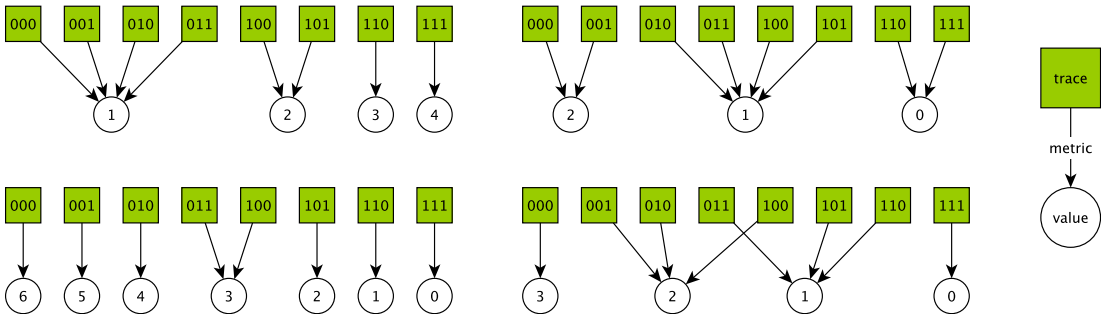

Figure 2.8: Four example metrics showing mappings of traces to values. The values are determined as follows (clockwise from top left): 1) number of samples until a zero is found. 2) number of zeros in first two samples. 3) number of zeros in the whole trace. 4) value based on how many zeros are found, and how quickly they are found.

Theorem 18 (RNFL)**.** *Let* β *be a set of test functions,* $\beta = \{f_1, f_2, \ldots, f_m\}$*, let* $\mathcal{A}$ *be a set of optimisation algorithms,* $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$*, and let* $\mathcal{M}$ *be a set of optimisation metrics,* $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$*. Then there exists a "restricted set"* $R_{\mathcal{A},\mathcal{M}}(\beta)$*, with* $\beta \subseteq R_{\mathcal{A},\mathcal{M}}(\beta) \subseteq F_{\mathcal{A}}(\beta) \subseteq \beta_{CUP}$ *such that all algorithms in* $\mathcal{A}$ *have the same average performance over* $R_{\mathcal{A},\mathcal{M}}(\beta)$*. A restricted set* $R_{\mathcal{A},\mathcal{M}}(\beta)$ *always exists, regardless of the choice of* β*,* $\mathcal{A}$ *and* $\mathcal{M}$*. In some cases it is identical to the focus set* $F_{\mathcal{A}}(\beta)$ *from the FNFL Theorem, but it can also be strictly smaller than the focus set.*

Proof. We give a proof by providing an example in which the restricted set is smaller than the focused set. The fact that a restricted set always exists follows from the fact a focused set always exists, thus we need only to show that the restricted set is potentially a subset of the focused set. Let $X = \{1,2,3,4,5,6\}$, $Y = \{0,1\}$, $\beta = \{101001, 001011, 001111\}$, $\mathcal{A} = \{A_1, A_2\}$ where A_1 deterministically samples from left to right and A_2 deterministically samples $f(2), f(4), f(6), f(1), f(3), f(5)$ in that order. Finally, let $\mathcal{M} = \{m_1\}$ where m_1 just returns the number of samples until the first 1 is found.

If follows from the definitions of the algorithms that, when run on the functions in β, A_1 produces the traces $\{101001, 001011, 001111\}$ and A_2 produces the traces $\{001110, 001011, 011011\}$. These sets of traces are not equal, and when we put these sets of traces through the metric m_1 they result in the multi-set of values $\{1,3,3\}$ and $\{3,3,2\}$ respectively, which are also not equal.

If we define $R_{\mathcal{A},\mathcal{M}}(\beta) = \{101001, 001011, 001111, 010010\}$ then running A_1 on the functions in $R_{\mathcal{A},\mathcal{M}}(\beta)$ produces the traces $\{101001, 001011, 001111, 010010\}$ and similarly running A_2 produces the traces $\{001110, 001011, 011011, 100001\}$. These sets of traces are still not equal, thus $R_{\mathcal{A},\mathcal{M}}(\beta)$ is not the "focus set" from the FNFL. However, when we put these sets of traces through the metric m_1 they result in the same multi-set of values $\{1,2,3,3\}$. Thus, $R_{\mathcal{A},\mathcal{M}}(\beta)$ is a "restricted set" and we are done. See Figure 2.9 for a visualization of the proof.

□

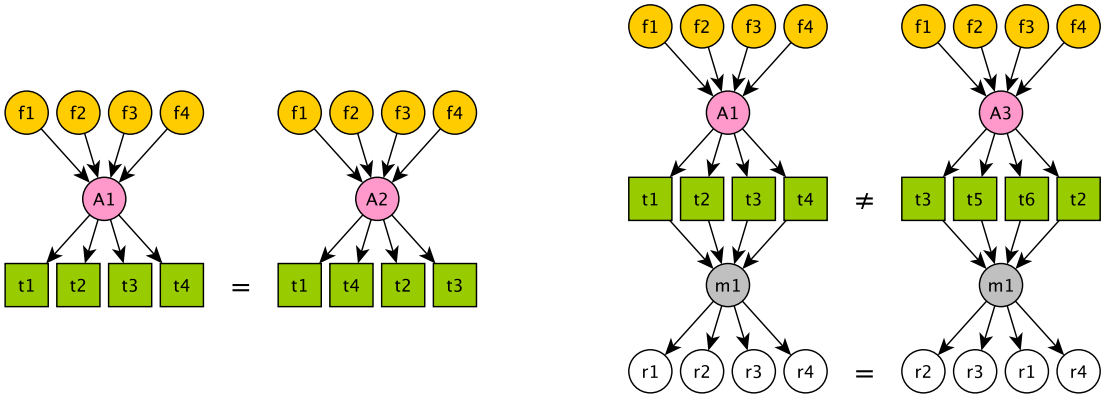

Figure 2.9: On the left is a visualization of a FNFL result for $\beta = \{f_1, f_2, f_3, f_4\}$ and $\mathcal{A} = \{A_1, A_2\}$. Over this set of four functions both optimisers produce the same set of traces, namely $\{t_1, t_2, t_3, t_4\}$. On the right we show that a RNFL can hold when a FNFL does not. For $\beta = \{f_1, f_2, f_3, f_4\}$ and $\mathcal{A} = \{A_1, A_3\}$ FNFL does not hold as A_1 produces the traces $\{t_1, t_2, t_3, t_4\}$ where as A_3 produces $\{t_2, t_3, t_5, t_6\}$. However, if we set $\mathcal{M} = \{m_1\}$ then a RNFL result holds, as both sets of traces lead to the same set of scores, $\{r_1, r_2, r_3, r_4\}$.

2.6.8 Multi-objective No Free Lunch

So far we have considered metrics that map to a scalar. However, within the optimisation literature there is much work on so-called multi-objective optimisation problems,

where the metric assigns a vector rather than a scalar. A natural question to ask is whether no free lunch results generalise to these situations. The answer is yes [27, 28]. The proof used in [27] works by defining a bijection between multi-objective problems and scalar problems. A further multi-objective result in [27] is that a no free lunch holds over the set of all multi-objective functions with any particular shape of Pareto front.

However, as they show in [29], in the case of multi-objective optimisation real world constraints (such as the algorithm only having finite memory) can readily result in algorithms with differing performances. The key idea is that, in scalar optimisation keeping track of your current best solution is straight forward, whereas in multi-objective optimisation problems where you are searching from the Pareto front it becomes more practically difficult to store the current best (in this case a set of points making up the Pareto front) efficiently, and so even fairly weak restrictions on the algorithm can mean that practically you need to instead store some sort of approximation of the Pareto front.

Thus, theoretically NFL holds for multi-objective functions, but when it comes to implementation multi-objective optimisers more often need to violate the NFL assumptions for reasons of pragmatism.

2.7 Interpretations and Unifying Results

We have now seen various no free lunch theorems and refinements. In this section we seek to provide some unification of the theorems and consider general approaches to optimisation that lead to the no free lunch results seeming most natural.

2.7.1 Optimisation as Interactive Induction

A standard computer science style phrasing of the problem of induction is the following: You receive a series of bits one by one. Each time you receive a bit you then use all of the bits you've received so far to try to guess whether the next bit will be a 0 or a 1.

Optimisation, in comparison, presents you at each point with the option of *which bit to look at next*, and your aim is no longer to accurately predict the values, but rather to use your (hopefully accurate) predictions to achieve some search metric specified goal. It is this choice of where to examine, as opposed to having always to simply

examine the *next* bit, that allows us to think of optimisation as an interactive version of the induction problem.

Once a formal relationship between optimisation and induction has been established, known results on induction can potentially be of use for optimisation researcher. To this end an induction phrasing of optimisation is perused by Everitt in [30], with the aim of understanding how Solomonoff's universal induction approach can be tweaked to apply to optimisation problems, and how, if so adjusted, the results would relate to no free lunch theorems. The conclusion is that the application of universal induction to optimisation does produce a small free lunch (by assuming problems are drawn from Solomonoff's universal prior), but the results are much weaker than in the sequence prediction setting. We discuss this free lunch in Section 2.8.3.

2.7.2 Probabilistic vs. Set-Theoretic No Free Lunch

There are two distinct but reconcilable framings of the optimisation setting for the no free lunch theorem. The first is a set theoretic approach, where we consider sets of problems to be solved but do not make any use of probabilities or expectations. In this scenario no free lunch results are of the form "the aggregate/average performance over all problems is algorithm independent."

The second approach is to consider an ensemble of possible problems. This problem ensemble is a probability distribution over the possible functions. In this setting no free lunch theorems take the form of assertions like "all optimisation algorithms have equal expected performance". This formulation is more powerful than the set theoretic approach as non-uniform distributions can be considered.

2.7.3 Geometric Inner-Product Interpretation

Another way to phrase the no free lunch theorem result is in terms of inner products. This interpretation is from Wolpert in [31], where he discusses "the inner product at the heart of all search". An inner product associates to pairs of vectors a scalar representing how aligned those vectors are. The inner product here indicates how aligned an optimiser is with a problem class. The geometric inner-product interpretation makes it somewhat intuitively obvious that an optimiser can be well aligned with some, but not all, problems.

2.7.4 Block Uniform Distributions

In [20] English presents an intuitive necessary and sufficient condition for NFL. He defined *block uniform* distributions, and proved that NFL holds if and only if functions are sampled from a block uniform distribution over functions. We state the theorem below, and a proof can be found in the paper.

Definition 17 (Block-uniform distribution)**.** *A probability distribution over the set of functions* $\{f : X \to Y\}$ *is block uniform iff* $\forall f, \forall \phi, P(f) = P(f_\phi)$*, where* ϕ *is a function permutation (see Definition 3, Chapter 1).*

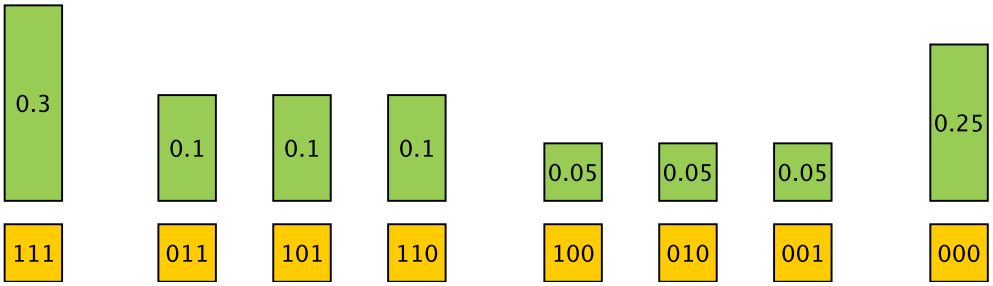

Figure 2.10: An example of a block uniform probability distribution over functions $f : \{1,2,3\} \to \{0,1\}$. There are four "blocks", corresponding to the four constituent permutation closed subsets of functions. The figure is not to scale.

Theorem 19 (NFL iff Block Uniform)**.** *For any metric, all optimisation algorithms have the same expected performance if and only if there is a block uniform probability distribution over functions.*

2.7.4.1 ε-Block Uniform

In [30] Everitt investigates what happens when a distribution is *almost* block uniform. He proves that the amount of free lunch available increases at most linearly with increasing ε, where ε characterizes the distance of a distribution from block-uniform.

Thus, not only is it the case that NFL iff block uniform, but also the distance of the probability distribution over functions from block uniform upper bounds the amount of free lunch available.

2.7.5 Fix Function, Pick Optimiser

There is an alternative presentation of the no free lunch result which can be surprising on first glance, even to those familiar with the traditional no free lunch statement. It can be seen as a backward no free lunch result. If you choose a particular function to be optimised and then uniform randomly pick an optimisation algorithm to use, you get a uniform distribution over possible traces, i.e. permutations of the selected input function. If you think of the standard NFL result as: for any optimiser, the results over all functions are equal, then this backward presentation says: for any function, the results over all optimisers are equal.

2.7.6 Stochasticity

From the very first papers on no free lunch theorems there has been consideration of their applicability to stochastic search algorithms. Although they may initially seem more powerful, and one may question the applicability of no free lunch results, in fact, all of the above results also apply to stochastic optimisation algorithms too.

In Theorem 11 we gave a proof that the original NFL theorem holds for stochastic optimisers. Similar proofs can be given for all the other NFL results we have seen still holding for stochastic optimisers. We consider stochastic optimisers in detail in section 4.5.

2.7.7 Parallelism

Another limitation on the NFL results above is that all the discussion has focused on algorithms that sequentially sample, evaluating points one-by-one. If multiple points are sampled in parallel do we get different results? This is a natural question to ask, especially given the increasing popularity of highly parallel architectures.

Theorem 20 (Parallel No Free Lunch)**.** *No free lunch theorems hold regardless of the number of samples you can take in parallel or the number of computations you can perform in parallel.*

Proof. An optimisation algorithm that evaluates n points in the function simultaneously is in fact *behaviourally more restricted* than a sequential algorithm (although it may well be *faster*). This is because it must decide the next n points to sample in one go, whereas the sequential algorithm only has to decide the next sample point, and can

then use the result of that function evaluation in the next sample point choice, and so on. □

Parallel algorithms can't behave more intelligently than their sequential competitors, but they may run faster, allowing them to do more in a given time. This chapter however focuses on algorithm behaviour, and in this regard, as we have seen, parallel algorithms are at best, equivalent to sequential algorithms.

2.8 Free Lunches

> *"while the NFL theorems have strong implications if one believes in a uniform distribution over optimization problems, in no sense should they be interpreted as advocating such a distribution."*
>
> — D. H. Wolpert 2012 [31].

As we have seen, no free lunch and related theorems are applicable in a wide range of situations. There are however various interesting scenarios where the results are not applicable. In this section we highlight settings in which the no free lunch theorems *do not apply*, an thus to a greater or lesser extent a "free lunch" is possible. The free lunches covered here are achieved by altering the set of functions considered, by considering suitable non-uniform probability distributions over functions, or by slightly altering one of the definitions used in the original theorem.

2.8.1 Bounded Description Length Free Lunch

The minimal description length of a function is, loosely speaking, the shortest program that maps $x \to f(x)$ for all $x \in X$. In 2003 Streeter showed in [32] that if we only consider functions $F = \{f \mid f : X \to Y\}$ with a minimal description length below a threshold θ for θ below an $|X|$ and $|Y|$ dependent constant, then some algorithms outperform others under some metrics when their performance is averaged over F. In other words, a free lunch is possible. This free lunch is a result of exploitable structure in F that results from the description length bound. Specifically, the restriction means that only a subset of functions are considered, and over these functions some optimisers outperform others.

2.8.2 Restricted Smoothness Free Lunch

If we introduce a metric on X and a metric on Y then we can define the smoothness of functions $f : X \to Y$. It has been shown that smoothness constraints can result in free lunches. In fact, although often not phrased as such, many theoretical assurances are examples of restricted smoothness free lunches. We consider two examples below, and direct readers toward [33] for general proofs that free lunches are available for cases where a maximum steepness of function or maximum number of local optima restriction is imposed.

2.8.2.1 Sub-Median Seeker

A classic example of a restricted smoothness free lunch is the sub-median seeker presented in [34]. In this paper Christensen and Oppacher endeavour to understand what makes a function searchable, in light of no free lunch. As they observe, "once [the no free lunch] theorem has been proven, the next logical step is to characterize how effective optimization can be under modest restrictions." In the paper they introduce the so-called Sub-Median Seeker algorithm, and then prove that it performs better than random search in a restricted-smoothness setting.

2.8.2.2 Gaussian Process Optimisation

We now provide an example of a restricted smoothness free lunch from an area of optimisation research where free lunches are regularly shown, but are generally not interpreted as such. There has been much work on the optimisation of functions drawn from a Gaussian process, and it has been variously shown that a suitable probability distribution over function smoothness is in fact enough to allow a free lunch [5, 6].

Note that a probability distribution over function smoothness is less restrictive than a restricted smoothness setting, as in the latter functions of below a certain level of smoothness are barred, whereas in the former setting they are just suitably unlikely.

2.8.3 Solomonoff-Levin Distribution Free Lunch

As we have disscused in Section 2.8.1, in [32] it is shown that a free lunch is possible for functions when the description length of the functions is sufficiently bounded. More generally, a free lunch is possible for functions if their probabilities are assigned according to a Solomonoff-Levin distribution. Under this distribution the probability

of a function is inversely related to length of its shortest representation, and it has been argued that this distribution, taken as a universal prior, is a formalisation of Occam's Razor. In the paper they also argue that Solomonoff-Levin type probability distributions are more relevant to optimisation than the block-uniform distributions that result in no free lunch results, as, roughly speaking, they are more realistic.

2.8.4 General Not Block-Uniform Free Lunch

In the bounded description length, restricted smoothness and Solomonoff-Levin cases the free lunches result from the exclusion of certain cost functions from the problem set, or suitably adjusting the probability distribution over the cost functions. (Note that excluding a function is equivalent to setting its probability to zero, so we can actually see all of these examples as probability distribution modifications).

In fact, we have seen in Section 2.7.4 that in all cases when the probability distribution over cost functions is not block uniform, some sort of free lunch is available. All the aforementioned results then are simply examples of this fact. We could list endless non block-uniform probability distributions over functions, each of which would necessarily result in a free lunch. The above examples are pertinent though as they represent restrictions that have been thought of as realistic for "real world problems".

2.8.5 Resampling Free Lunch

Another requirement for the no free lunch theorems is that resampling of points in the domain either doesn't occur or isn't considered in evaluation of the algorithm, such that an algorithm is only evaluated on its new point sampling behaviour. However as often observed, for example by Whitley in [2], many algorithms try to exploit current good solutions by focusing future search in their vicinity, which leads to an increase in the chance of resampling (assuming no book-keeping is employed specifically to avoid this), in comparison to random search. This leads to the observation that random search is expected to be better than many popular optimisers over all possible functions simply because it is less likely to resample points, and in this sense it achieves a free lunch.

More generally, the less an algorithm tends to resample the better, and the actual behaviour of an algorithm that resamples less is expected to be better than the actual behaviour of an algorithm that resamples more. Note that this is different from the exploration behaviour, which only considers the way in which algorithms explore new

points, and with respect to which the free lunch theorems hold.

Simply put, when you ignore resampling all algorithms are equivalent in the free lunch sense, and when you do not ignore resampling then the less an algorithm resamples, the better.

2.8.6 Coevolutionary Free Lunch

One key assumption made in the no free lunch theorem is that the metric does not depend on the test function f. The necessity of this requirement is exemplified by the existence of Coevolutionary Free Lunches [35]. A coevolutionary free lunch is an example of a situation where free lunches do exist as a direct result of the metric used depending on f. Co-evolution can be seen as an example of self-play, and "in the self-play domain there are algorithms which are superior to other algorithms for *all* problems" [35].

This free lunch is particularly interesting because in nature, for example, it is not usually the case that some external fixed fitness metric is imposed from above, but rather fitness metrics emerge in some complex way from the dynamics of the system as a whole. This said however, the authors in [35] observe that real evolution in nature does not meet the the specific requirements for this free lunch to apply.

2.8.7 Minimax "Free Lunch"

In this section we examine whether algorithms can outperform one-another under minimax distinctions. Two algorithms can be minimax compared. To do this you compare the results of the algorithms on each function, and count how many times each algorithm has the minimum cost (i.e. the lowest of the two). The winning algorithm is the one of the pair that more often has the minimum cost.

As Wolpert theorised might be possible in [7] and Radcliffe and Surry gave an explicit simple example of in [22], some algorithms beat others in the minimax sense over all functions. The example from Radcliffe and Surry is shown in figure 2.11. However, importantly this outperforming is not transitive, and thus does not induce an ordering on the optimisation algorithms.

We argue here that this minimax domination is a free lunch only in the same way that one algorithm doing better than another on a particular function is a free lunch. Specifically, we prove that all algorithms minimax dominate and are dominated by exactly the same number of other algorithms.

	time to min					
f	A	B	C	A vs B	B vs C	C vs A
000	1	1	1	tie	tie	tie
001	1	1	2	tie	B	A
010	1	2	1	A	C	tie
011	1	3	2	A	C	A
100	2	1	1	B	tie	C
101	2	1	3	B	B	A
110	3	2	1	B	C	C
111	1	1	1	tie	tie	tie
winner				B	C	A

Figure 2.11: Comparison of algorithms A, B and C over functions mapping $\{1,2,3\}$ to $\{0,1\}$. Each algorithms simply samples the points in a given fixed order. Note that the total cost for each algorithm is 12 - they are all equal as implied by the no free lunch theorem. However, there is also non-transative minimax domination. This table is reproduced from [22].

Theorem 21 (Minimax Domination is Cyclic and Equally Distributed)**.** *Every optimisation algorithm minimax dominates the same number of optimisation algorithms, and every optimisation algorithm is minimax dominated by the same number of optimisation algorithms.*

Proof. If C minimax dominates A then an optimiser B can be constructed as $B = C \circ A^{-1} \circ C$, which will dominate C exactly as C dominates A. This construction is shown in Figure 2.12. In this way for any optimiser C, for every A_i that it dominates a corresponding B_i is defined that dominates C. It follows that every optimiser dominates the same number of optimisers as it is dominated by. It remains to show that this number is the same for all optimisers. However, this is a direct result of the fact that all the optimisers produce the same set of traces. □

2.9 No Free Lunch Discussion

We have presented the original no free lunch theorems, numerous extensions and considered various examples of situations in which free lunches can be achieved. We now

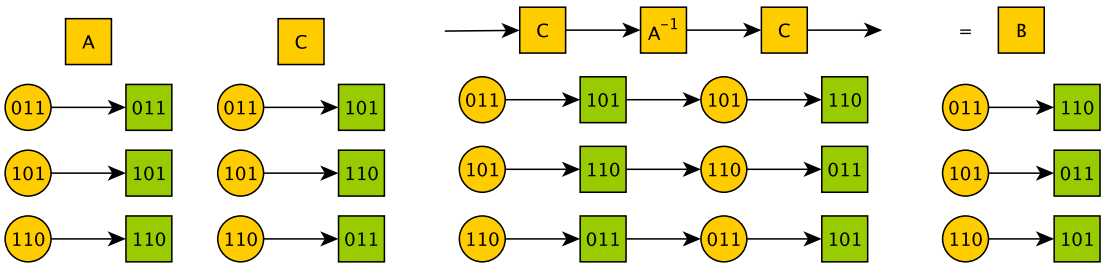

Figure 2.12: An example of the construction used in the minimax proof. The figure only shows a CUP subset of functions, but the same method works for the whole function set. Here A dominates C, and we can thus use C and A^{-1} to produce a B that dominates A in the same way.

conclude our examination of free lunches with some discussion.

2.9.1 Are No Free Lunch's Assumptions Realistic?

There has been much discussion about how realistic the assumptions of the no free lunch theorems are. In particular, do they pertain in the real problems dealt with in real-world applications? The consensus is that in the most straight forward sense the answer is no [36], and refinements such as SNFL can be seen as efforts to address this. This said, in Section 2.9.8 we argue that the assumptions required for no free lunch to hold do not have to be realistic for the theorems to be of interest.

2.9.2 Free Appetizers

Various researchers have discussed the idea that in realistic search situations the advantage a good algorithm has is small. This small advantage held by a well suited algorithm is sometimes called a "free appetizer". [36].

For example, a restricted complexity assumption on problems results in a probability distribution that breaks the requirements for no free lunch to hold, however even though a free lunch exists, it is at best only small, a so-called free appetizer.

Moreover, it has been shown that if we adopt the universal prior (a probabilistic version of the complexity restriction, where complex functions are not impossible, but just unlikely) then a free lunch can be achieved. However, this free lunch is again small, and the algorithm that obtains it is not computable [37].

Thus, even if there is some structure on the distribution of real problems that violates the no free lunch assumptions, the resulting free lunch may well be only minimal.

A free lunch isn't necessarily much of an improvement.

2.9.3 Computational Complexity

Complexity research also makes claims about the general solubility of problems, for example that some problems cannot be solved in polynomial time by a deterministic Turing Machine. There has been investigation into how these two areas relate and, it is generally the case that algorithmic complexity theorems put much tighter restrictions on optimisation algorithms than no free lunch theorems do [15]. As observed in [38], "No Free Lunch has not been proven to hold over the set of problems in the complexity class *NP*, ... in fact, proofs concerning No Free Lunch do not apply to *NP*-complete problems unless the proofs also show (perhaps implicitly) that $P \neq NP$."

2.9.4 Universal Bias

There has been an argument put forward that a so-called universal bias exists, and that this would allow a single algorithm to be successful in all real problems of interest [16]. This comes about by asserting that there is a pervasive bias towards problems with more structure, and thus the no free lunch theorem's requirement of a (block-) uniform distribution over functions is unmet. The idea is that if the intuitive concept of "structure" is made rigorous through Kolmogorov complexity then this bias towards problems with more structure is just the formalisation of Occam's razor. As we have seen in Section 2.9.2, Everitt in [37] shows that in fact the universal bias would only lead to a free appetizer type situation, in which a small free lunch could be obtained. See Section 2.8.3 for more discussion of the free lunch possibilities under a universal bias.

2.9.5 Algorithmic Complexity

NFL makes a statement about the general solubility of problems. In [39] Borenstein and Poli examine the relationship between a search problem's Kolmogorov complexity and its difficulty. They conclude that algorithmically complex problems are difficult but that the converse doesn't hold, that is, difficult problems are not necessarily algorithmically complex.

English proves a relationship between the complexity of an algorithm, the function input and the trace output in [19]. In particular he shows that the size of the difference

in complexity between the input and the resulting traces is bounded by the complexity of the optimiser itself. In the next chapter we will give more consideration to optimisation and algorithmic complexity.

2.9.6 No Free Lunch and Compression

An analogous result to the no free lunch theorem is the impossibility of a universal compression algorithm. It follows from the fact that there are more binary strings of length $n+1$ than there are binary strings of length n or less that any non-lossy compressor cannot compress every input. The compressor, to be most effective on average, must exploit the expected structure of the inputs, highly compressing common patterns at the expense of only weakly compressing or even expanding rare patterns. This is the idea behind Huffman coding.

Just as a compression algorithm cannot universally compress but must exploit expected input structure to compress well on average, so an optimisation algorithm cannot work well on all possible inputs, but must exploit expected input structure to optimise well on average. As we will see the Bayesian interpretation we put forward in the coming chapters will directly relate behaviour to what problems the optimiser expects.

2.9.7 Bayesian No Free Lunch

Roughly speaking, in a Bayesian framing of optimisation the standard no free lunch theorem becomes the claim that informative induction can't be done without prior assumptions. In fact, this is a widely known maxim, the necessity of an inductive bias voiced by various researchers: "A basic insight of machine learning is that prior knowledge is a necessary requirement for successful learning" [40], in other words, "[Y]ou can't do inference ... without making assumptions" [41].

Making this relationship more precise, Streeter has shown in [32] that the no free lunch result applies only when a certain form of Bayesian learning is impossible. More specifically, they prove that a no free lunch result holds if and only if for any $D \in \mathcal{D}$ and $y \in Y$

$$P(f(x_i) = y|D) = P(f(x_j) = y|D) \; \forall \, x_i, x_j \in X \setminus D_x$$

In other words, regardless of what data you have so far gathered about the function, you are still unable to in any way distinguish the unsampled points. (Note that $X \setminus D_x$ is the set of so far unsampled X values, i.e. those that do not yet appear in the data).

In 2007 Carroll and Seppi examined how graphical models of supervised learning problems can be used to better understand the idea of optimal optimisers and no free lunch. They focus on understanding bias, meta-bias, transfer learning and over-fitting in a Bayesian way, and reconciling these concepts with the known know free lunch results. They conclude by proposing another Bayesian restating of the no free lunch result, that "a uniform prior over functions lead to a uniform off training set posterior over classes" [42].

More recently Serafino has been emphasising the important close relationship between Bayesian optimisation and no free lunch results [17, 43]. We think that these approaches are in the right direction, but they do not consider the fact that many optimisers are apparently not Bayesian. That is in essence what we try to address in this thesis.

2.9.8 Critique of No Free Lunch

Perhaps the most standard criticism of the no free lunch theorem is that its usefulness is limited by the fact that in reality *"the fit functions belong to a narrow subfamily of functions (e.g., smooth functions), and some algorithms work better than others on such families"* [44]. However, Wolpert himself emphasises in [31] that the original no free lunch theorem does not advocate a uniform distribution over optimisation problems, and quotes his original paper which states that the no free lunch theorems tell us about "the underlying mathematical 'skeleton' of optimization theory before the 'flesh' of the probability distribution of a particular context and set of optimization problems are imposed" [8].

Essentially, the mistake of the above criticism is to interpret no free lunch as a theorem aiming to show that all optimisers are equivalent, but relying on false assumptions. The reality is that it serves as a proof of the fact that in order to compare optimisers you need to know something about the sorts of problems they will be dealing with. Moreover, as a result of the no free lunch theorem it is now generally appreciated by researchers that no optimisers are simply behaviourally better than others a-priori, but that knowledge of the context of their use is vital.

2.9.9 Order Statistics

The no free lunch theorem states that all optimisers perform as well as random search when averaged over all functions. A natural next question then is: "how well does

random search do?". This is a question that receives surprisingly little attention in the literature, considering the number of times the first observation is made. In order to address the question we make use of known order statistics results.

Assume a uniform distribution over functions $f : X \to Y$. It follows that when sampling a function f from the distribution and then evaluating f at any x we have a uniform distribution over expected Y values. Also, it is the case that for all $x_1, x_2 \in X$, if $x_1 \neq x_2$ then $f(x_1)$ and $f(x_2)$ are independent. Using this knowledge we can calculate the expected minimum Y value found after k samples using the order statistics of a multinomial distribution with $|Y| = m$ possible outcomes.

Assume without loss of generality that $Y = \{\frac{0}{m-1}, \frac{1}{m-1}, \ldots, \frac{m-1}{m-1}\}$, so that for any m the values in Y are uniformly spaced over the unit interval. Let S_k be the set of y values seen after k samples. We now examine the $0th$ order statistic, that is, the minimum value in S_k, $\min(S_k)$, with respect to m and k. The probability that $min(S_k)$ takes a particular value is:

$$P\left(\min(S_k) = \frac{x}{m-1}\right) =$$

$$\sum_{j=0}^{k-1} \binom{k}{j} \left(\left(\frac{m-1-x}{m}\right)^j \left(\frac{x+1}{m}\right)^{k-j} - \left(\frac{m-x}{m}\right)^j \left(\frac{x}{m}\right)^{k-j} \right) =$$

$$\frac{(m-x)^k - (m-x-1)^k}{m^k}$$

Thus, the expected value, as a function of m and k is:

$$\mathbb{E}[\min(S_k)] = \sum_{x=0}^{m-1} \left(\frac{x}{m-1}\right) \left(\frac{(m-x)^k - (m-x-1)^k}{m^k}\right) = \sum_{x=0}^{m-1} \frac{x^k}{m^k(m-1)}$$

If we define U_k as k samples from a *continuous* uniform distribution on the unit interval (as a posed to the multinomial above) it is a known result that:

$$\mathbb{E}[\min(U_k)] \to \frac{1}{k+1}$$

Thus, in both the multinomial and the continuous case the minimum quickly approaches the minimum value in Y. It seems then that if we assume no structure, all optimisers perform well at optimisation.

In fact, these results have an interesting additional interpretation, they show that there is no curse of dimensionality in unstructured problems, as the expected sample

minimum is independent of the domain size $|X|$. That is, in the above problem setting, the progress we expect to make in finding the minimum isn't influenced by the size of X (assuming we do not intend to eventually sample every $x \in X$).

On the other had, for a very structured space of problems, by the very nature of having such structure, the curse of dimensionality is again non-problematic, as the search decisions can be strongly informed by the knowledge of the underlying structure and the samples seen so far.

Thus, it seems that there should be an intermediate degree of structure at which a search problems is, to continue the metaphor, *maximally cursed*.

Here we have worked under the assumption that all functions are equally likely, and thus the distribution of y-values is uniform. However, similar calculations can be worked through for alternative distributions of y-values resulting from different probability distributions over potential functions. In each case the zeroth order statistic would tell us the expected performance of random search (for minimisation), and any exploitation of local or global problem structure by an optimiser would improve on this lower bound.

If there was no exploitable structure then no optimiser would be expected to do better or worse than the zeroth order statistic. If they do better, this necessarily means there must exist some exploitable structure, and the better they do, the more there is. This leads to the following order statistic framing of the no free lunch result:

Theorem 22 (Order Statistic No Free Lunch Formulation)**.** *A no free lunch holds if and only if no optimiser can do better (on average) than the zeroth order statistic for the appropriate y-value distribution. More formally, let P_y be the distribution over Y values resulting from random search. Let z_i be the zeroth order statistic of P_y after i samples. Let $E_A[y_i]$ be the expected value of the minimum y value found by optimiser A after i samples. The original no free lunch theorem holds if and only if, for all A, $z_i = E_A[y_i]$.*

Proof. Follows directly from the original no free lunch theorem. See the above discussion for details. Intuitively, the zeorth order statistic z_i gives the expected performance of random search, and the no free lunch theorem holds if and only if no stratergy is better or worse than random search, i.e. if and only if $z_i = E_A[y_i]$ for all A. □

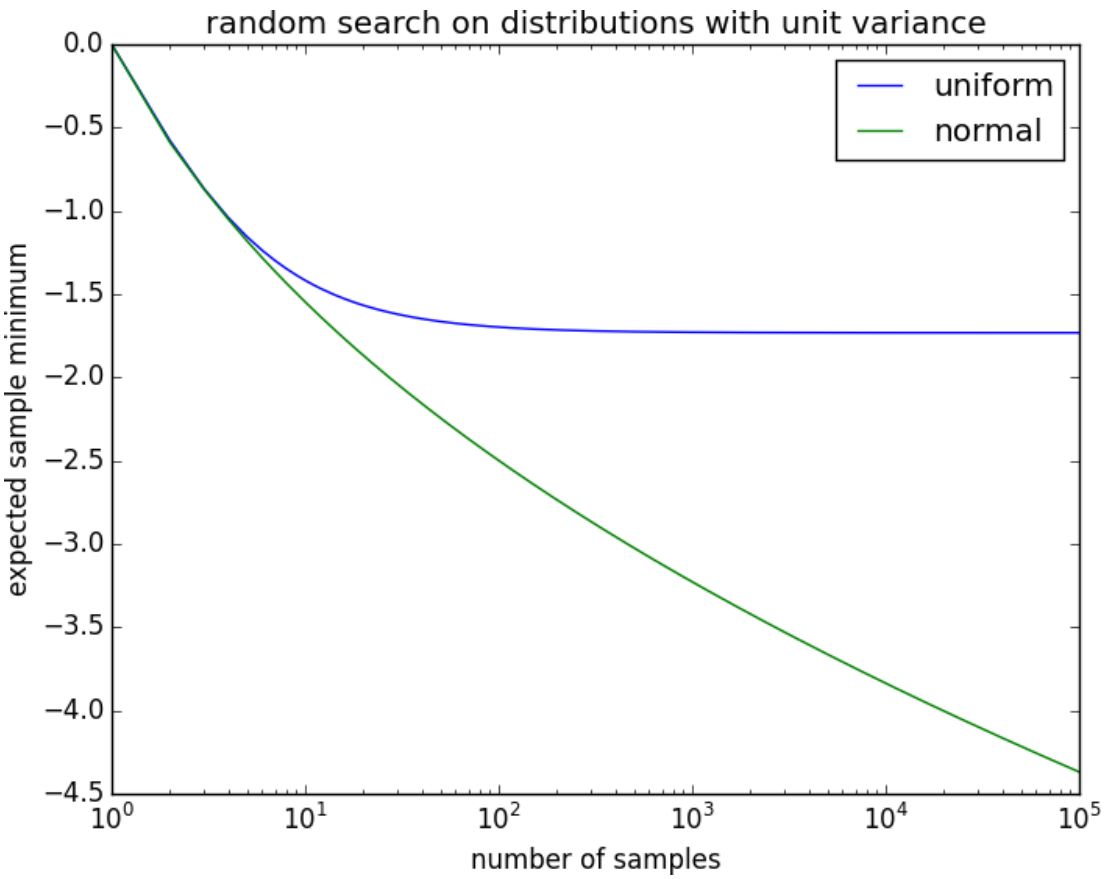

Figure 2.13: The expected minimum found using a random search on a uniform and a Gaussian distribution of Y values, both scaled to have unit variance and shifted to have zero mean.

2.9.10 Needle in a Haystack

The above reasoning leads to the following conclusion: when no free lunch holds, random search (and thus any optimisation algorithm) is expected to do well. Is it possible to construct a no free lunch scenario without this property? The answer is that we can construct the hardest possible optimization problem, and show that this converges slowly.

Consider the set of functions that are zero everywhere except for at a single X value at which they are one. These are sometimes called needle-in-a-haystack functions. For these functions, the expected reward (or equivalently the probability we have found the "needle") after k samples when $|Y| = m$ can be seen to be $\frac{k}{m}$, or $1 - (\frac{m-1}{m})^k$ if we allow resampling.

2.9.11 Infinite and Continuous Lunches

In this exploration of the no free lunch theorem we have not considered infinite domains and continuous extensions, though work to this end can be found in the literature [45, 46, 47]. This decision is in part due to the fact that the algorithms in which we are interested, and will be exploring in detail in the coming chapters, will be running on finite sets of possible inputs, and in part because optimisation algorithms in general use run on finite procession machines.

2.9.12 Other No Free Lunch Results

Here we have focused on no free lunch in search and optimisation, but there are no free lunch type theorems know in other areas too, for example in supervised learning [48], cross validation [49] and human-machine interface [50]. We also have not discussed time varying cost functions, which are examined in [7].

2.10 Comparing Optimisers After No Free Lunch

In the above sections we have covered in detail the original no free lunch theorems, their extensions and modifications and their caveats and exceptions. It is evident that, as emphasised in [51], the existence and importance of free lunches is far from a straight forward question, and the opinions of researchers vary. This said, we now try to briefly summarise what the results, when considered as a whole, seem to really mean for algorithm comparison and evaluation:

1. The no free lunch results preclude meaningful comparison of optimisation algorithm's *exploration behaviour* without reference to specific problems.

2. However, almost all restrictions on the set of problem functions result in possible free lunches.

3. Similarly, but more generally, almost all probability distributions over problem functions result in possible free lunches.

4. More specifically, block-uniform distributions capture exactly the scenarios where no free lunch results hold for any metric.

5. However, when we are interested in no free lunch results with respect to particular metrics, and for limited numbers of samples, then free lunches are possible even under block-uniform distributions.

6. When free lunches are possible, their prominence tends to depend crucially on the optimisation metric used.

7. When free lunches are possible, the algorithms that achieve them are *aligned* with the probability distribution over problem functions (what exactly this means is discussed in the next chapter).

8. When considering more than just the exploration behaviour of an algorithm, algorithms can be ranked. For example, some optimisers are simpler, some are faster and some tend to resample less than others.

Pragmatically the results mean that benchmarking alone cannot be used to evaluate an algorithm, but must be used in combination with clear underlying assumptions. The benchmark functions must be representative of the problems and there must be some smoothness, in the sense that being good at a problem means that you are likely to be good at similar problems.

Of particular importance for this thesis is the idea that the *alignment* of problem structure and algorithm behaviour determines how well an algorithm performs on the problem. This statement is well captured by the inner-product phrasing of the no free lunch theorem [31] (see section 2.7.3). In the following chapters we try to understand how this idea of alignment can be best rigorously characterised.

Chapter 3

Bayesian Optimisation

3.1 Introduction

In the previous chapter we saw that no free lunch results mean that we should not search for fully general optimisers, as these do not exist, and we should instead focus on producing optimisation algorithms that are behaviourally *aligned* with the problems we want to solve. In this chapter we begin to investigate more closely this idea of alignment. We will look at what it means for an optimisation algorithm to be Bayesian, and show that a Bayesian approach to optimisation allows us to make this problem of alignment rigorous.

Using this Bayesian approach we then present a general method for interpreting the behaviour of optimisation algorithms. We show that all deterministic function optimisation algorithms behave as Bayesian optimal greedy optimisation for some prior, and simultaneously as optimal information gain optimisation for another prior. We also show that, in both cases, the Kolmogorov complexity of the prior is equal to the Kolmogorov complexity of the original optimisation algorithm up to a constant.

There has much been recent interest in Bayesian optimisation, and it has proven to be as good as or better than the state of the art in many optimisation tasks [52]. The standard view is that Bayesian methods differ from other optimisation techniques in that they construct a probabilistic model of the function being optimised and use this model to inform their decisions [53, 54]. Often, for example, a Gaussian process is used to define the prior distribution over functions, as it is both flexible and computationally tractable [55, 54].

The statements are formulated in this chapter for a deterministic, discrete problem, and in later chapters we relax these constraints. Some discussions on the relaxations

to follow is given in section 3.10.

This chapter proceeds as follows: In Section 3.2 we define Bayesian optimisation, in Section 3.3 we give some chapter specific preliminaries, in Section 3.4 we make the concept of alignment rigorous, in Section 3.5 we present in detail a proof that all deterministic optimiser's on-policy behaviour can be seen as Bayesian, in Section 3.7 we give our complexity argument, in Section 3.8 we discuss the idea of simultaneous interpretation and we conclude in Section 3.10 with more general discussion.

3.2 Bayesian Optimisation

Thomas Bayes (1701-1761) was an English statistician, philosopher and Presbyterian minister. He is most famous for his eponymous theorem, which describes how probabilistic beliefs should be updated in light of evidence.

In a Bayesian setting we have a probability distribution P over $\mathcal{F}$, called the prior, which we combine with data $D \in \mathcal{D}$ using Bayes' theorem to yield a posterior.

$$P(f|D) \propto P(D|f)P(f)$$

As we are dealing with noise-free optimisation the likelihood $P(D|f)$ is an indicator function that is zero everywhere except where D agrees with f.

$$P(D|f) = 1 \Leftrightarrow \forall\, (x,y) \in D, f(x) = y$$

The posterior is then used in conjunction with a cost function to choose the next sample point. Thus, in order to define a Bayesian optimisation algorithm all that is required is to specify the prior and the cost function. In this setting a cost function b is of the form $b : X \times \mathcal{P} \to \mathbb{R}$.

Having specified the prior P and cost function b a sampling policy $s : \mathcal{D} \to X$ can be constructed as follows.

$$s(D) = \operatorname*{argmin}_{x \in X} b(x, P(\cdot\,|D)) \tag{3.1}$$

Various functions can be used as the cost function b. A simple example is the expected value of $f(x)$ given P.

$$b(x,P) = \sum_{f \in \mathcal{F}} P(f) f(x) \tag{3.2}$$

For this choice of cost function, b, the resulting sampling policy samples the point at which it expects the lowest output will be obtained. This is a simple greedy policy for function minimisation. We define s_g to be this sampling policy, that is:

$$s_g(D) = \operatorname*{argmin}_{x \in X} \sum_{f \in \mathcal{F}} P(f|D) f(x) \tag{3.3}$$

The simple greedy optimiser s_g will be used throughout the remainder of this chapter as an example Bayesian optimisation algorithm.

3.3 Preliminaries

In this section we introduce the definitions and notation used throughout the chapter.

3.3.1 Complexity

Measuring complexity is a difficult and ill-defined task. In this chapter we use Kolmogorov complexity (also called algorithmic complexity), which we simply call complexity. This is a common choice in the literature and is well theoretically grounded [56]. Here we give an intuitive definition of this complexity measure.

Definition 18 (Kolmogorov Complexity)**.** *The complexity of a string s can be intuitively thought of as the length in bits of the shortest program that produces s as an output and then halts. We write $K(s)$ to mean the complexity of s.*

An important fact is that the complexity of s is programming language independent up to a language dependent constant factor. This means that for two programming languages p_1 and p_2, if we write $K_{p_i}(s)$ to mean the complexity of s using language p_i then we know that $K_{p_1}(s) - c_1 \leq K_{p_2}(s) \leq K_{p_1}(s) + c_2$ for two constants c_1 and c_2 that do not depend on s.

3.3.2 Notation

In this chapter, without loss of generality, we will consider function *minimisation* algorithms. Thus the algorithm's aim to find a minimum of a function $f : X \to Y$ for some finite domain X and range $Y \subset \mathbb{R}$. Again we fix X and Y and let $\mathcal{F}$ be the set of all such functions, $\mathcal{F} = Y^X$, as we have done in previous chapters.

As discussed above, here we look at function optimisation from a Bayesian perspective. Thus, we will be concerned with priors. In this setting a prior is a probability distribution over $\mathcal{F}$. We define $\mathcal{P}$ to be the set of all such probability distributions and let $P \in \mathcal{P}$ to be some arbitrary prior.

3.3.3 On-Policy Behaviour

For the remainder of this chapter we will only be concerned with the on-policy behaviour of optimisation algorithms, and two such algorithms are thought of as equivalent if they have identical on-policy behaviour. When we say two algorithms are the same, we mean only that they have the same on-policy behaviour.

3.4 Alignment

So far we have claimed that the only way one optimisation algorithm can be better behaviourally than another is by being more *aligned* with the problem. This claim is the result of examination of the no free lunch theorems in the previous chapter. But what exactly is alignment? In this section we examine alignment, try to make its meaning clear, and argue that it is best understood in a Bayesian way, that is, through the idea of exploiting prior beliefs about the problems likely to be encountered.

Let us start by assuming that we are interested in optimising functions of some particular kind. For example, we could be interested in how the quantities of various chemicals present in the environment affect cell growth, with the aim being to find the mixture of chemicals producing the fastest growth. Thus, phrased as an optimisation problem, the aim would be to find the maximum of some function $f : X \to Y$ where $x \in X$ represents the mixture of chemicals used and $y \in Y$ is the resulting cell growth. Alternatively we could be concerned with a very broad range of functions, for example we might simply be interested in all computable functions.

Regardless of the kind of problems we focus on (the problem class), the assumption is that we can think of the functions representing problems in this class as being drawn from some probability distribution over all functions. We call this probability distribution the *real problem distribution*. This distribution exists and is fixed, although the details of it are most likely unknown. A problem class then is simply a probability distribution over functions.

Alignment is intended to be a way of describing how well a particular optimisation

algorithm is suited to a particular problem class (i.e. a particular probability distribution over functions). Intuitively, an algorithm and a problem class are aligned if the algorithm usually does well on problem instances from that class. The better an optimiser typically performs on a problem, the more aligned it and the problem class are.

If we were using Bayesian optimisation, and we *did* know the real problem distribution, then regardless of our optimisation metric (e.g. find the minimum) we would be able to optimally sample the function, achieving our aim with maximum efficiency.

However, in general we do not know the real problem distribution, and instead, if we want to perform Bayesian optimisation, we have to make an educated guess. We behave in accordance with our guess of the distribution, and hope that this is close to the optimal behaviour, which is exactly the behaviour that would result if our guess was correct.

It is not straight forward to calculate how errors in our guess at the real problem distribution translate to deviation in our behaviour from the optimal. Firstly, one can have a guessed problem distribution that is different from the real problem distribution, but still results in optimal behaviour. Secondly, the way in which discrepancy between the real and guessed problem distributions translates into a difference between the actual and optimal behaviour is dependent on the optimisation metric. One way to capture this second fact is to say that the distance metric on the space of probability distributions used is determined by the optimisation metric being used in the optimisation. In this case the closeness of probability distributions is dependent on the similarity of the behaviour they produce.

These ideas allow us to be rigorous about alignment in a Bayesian optimisation setting. An optimisation algorithm and problem class are aligned if the probability function that describes the problem class is close to the optimiser's prior distribution with respect to the metric induced by the optimiser's cost function. Here what we mean by close depends on what we want to mean by aligned. The closer the two distributions are in the above sense the more aligned the optimiser and problem class are.

3.5 All Deterministic Optimisers are Bayesian

This section contains the first of the main results in this chapter. We have so far seen that, as the no free lunch theorem makes explicit, no algorithm is behaviourally better than any other a-priori, and that in order to do well at a particular type of problem an

algorithm must be *aligned* with the structure of that problem. In Section 3.4 above we argued that this concept of alignment is best captured in a Bayesian framework, in which the degree of alignment can be understood as the similarity between the probability distribution from which the function to be optimised really comes, the so called true problem distribution, and the Bayesian optimiser's prior. Thus, a Bayesian optimisation algorithm performs best if the prior it is behaving in accordance with matches the true problem distribution.

However, very many prominent optimisation algorithms are seemingly not Bayesian. The question then is what does alignment mean in these cases? The answer is that, *although the algorithms are not explicitly Bayesian, they are still Bayesian*. More specifically, they behave identically to some explicitly Bayesian optimiser. Because of this, the alignment of these algorithms with problem classes is defined in the same way, with the only difference being that rather than being able to look at the prior explicitly defined in the algorithm as you can in the Bayesian optimiser case, now you have to refer to the prior *implied* by the algorithm's behaviour.

Although the above is true for all algorithms, showing it will take most of the remainder of this thesis. In this section we make a start by showing how the on-policy behaviour of a prior and cost function pair can be mapped to the on-policy behaviour of a deterministic optimisation algorithm, and vice versa.

3.5.1 Prior and Cost Function $\rightarrow$ Algorithm

Before we look at how to decompose an algorithm into a prior and a cost function we first look at how a prior and a cost function can be composed to make an optimiser. If we are tasked with constructing an optimisation algorithm from a given prior P and cost function b it is simply a case of assembling the pieces as described in section 3.2. The sampling policy A can be directly constructed as follows:

$$A(D) = \operatorname*{argmin}_{x \in X} b(x, P(\cdot|D)) \tag{3.4}$$

In this way A is produced directly from P and b. Whenever we refer to "the algorithm resulting from the cost function and prior", or similar, we mean an algorithm, A, produced by inputting the relevant cost function and prior into equation (3.4).

3.5.2 Algorithm $\rightarrow$ Prior and Cost Function

We now address the core of the problem. Given any arbitrary optimisation algorithm A, can we decompose it into a prior P and cost function b such that the algorithm defined through equation (3.4) by P and b has identical on-policy behaviour to A.

The mapping in this direction is more involved. One reason for this is that, as we will soon show, various prior and cost function pairs can result in the same algorithm. In other words the decomposition is not unique. Here we give two examples for an arbitrary sampling policy $A \in \mathcal{A}$. In the first example essentially all the complexity of the behaviour is embedded in the cost function, and in the second example the complexity is contained in the prior.

3.5.2.1 Complexity in the Cost Function

This example serves to show that various cost function and prior pairs can result in the same algorithm. Let $A \in \mathcal{A}$ be an arbitrary sampling policy, that is, $A : D \rightarrow X$. We use A to define the outputs of our cost function, regardless of the prior, as $b(x, P(\cdot|D)) = 0$ iff $A(D) = x$ and $b(x, P(\cdot|D)) = 1$ otherwise. As the prior has no effect on the cost we leave it unspecified.

Although this construction may initially seem circular, it is not. We do not need to use A to actually compute the values, but instead we just construct b to mimic the known behaviour of A, for example using a look up table mapping $\mathcal{D} \rightarrow X$.

This example can be thought of as making the cost function do all the work. The complexity of the prior could be made very small, and thus the bulk of A's complexity must be contained within the cost function b.

Although this is a legitimate decomposition, it isn't a very satisfying one. This cost function could just be anything and it seems reasonable to wonder why it is this way rather than that. Of course it is defined so as to produce the same behaviour as the original optimiser A, but it doesn't seem to do anything to help understand or interpret the original optimiser's decision. Essentially this decomposition hasn't aided our understanding of the algorithm. We are in no better position to say what sorts of problem the algorithm would excel on.

3.5.2.2 Complexity in the Prior

We now consider a decomposition that *is* helpful in understanding the algorithm. We first fix the cost function b, which can be seen as deciding what it is that the algorithm

is trying to achieve. Here we set b to be the expected value as defined in (3.2). Now we have what the algorithm is trying to achieve (captured in b), and also what the algorithm does in ever situation (captured in A). We can now deduce the prior belief P by iterating through all possible inputs to sampling policy A and noting the results.

The intuition is as follows: a sampling policy is nothing more than the decisions it makes in every situation. We can theoretically iterate through all possible situations and note the output of the sampling policy in each case, we can then build a prior from scratch that produces identical outputs to the ones observed when combined with the greedy expectation maximizing cost function, b.

Essentially we build a prior that, when conditioned on some data D produces a posterior in which the input expected to yield the minimum output is exactly the input selected to be sampled next by the algorithm when given the same input D, i.e. $A(D)$. Our cost function b is of some small fixed complexity and thus the bulk of the complexity must be captured in P.

We now prove that it is indeed possible to build a suitable prior for any sampling policy. Note that the reverse is trivially true, as shown in Section 3.5.1.

3.5.3 All Optimisers are Greedy

We have seen that the decomposition of an optimiser is possible both in the somewhat trivial way detailed in section 3.5.2.1 and the potentially more interesting way shown in section 3.5.2.2.

The claim that it is possible to construct a suitable prior for any sampling policy is made explicit in the following theorem:

Theorem 23 (All Optimisers are Greedy)**.** *For any optimisation algorithm, A, there exists a prior, P, such that, in any on policy situation, the x value sampled by A is exactly the x with minimum expected y* $(= f(x))$ *value according to P. That is,*

$$\forall A \in \mathcal{A}, \exists P \in \mathcal{P}\, s.t.\, D \in \mathcal{D}_A \Rightarrow A(D) = \underset{x \in X}{\operatorname{argmin}} \sum_{f \in \mathcal{F}} P(f|D) f(x)$$

In other words, all optimisers behave as if they are just greedily minimising with respect to a certain prior.

Proof. Assume we have some sampling policy A from which we want to determine a prior P. The cost function used in the theorem is exactly B_g from equation (3.3), and so at each point we will sample the x expected to give the lowest output $y = f(x)$.

Sampling policy A chooses the initial sample point x_0 based on no data. To capture this in the prior define $\bar{y} = \sum_{y \in Y} \frac{y}{|Y|}$. Set $P(f) = \alpha$ if $f(x_0) < \bar{y}$ and $P(f) = \beta$ otherwise for some $\alpha > \beta$. Thus $\mathbb{E}[f(x_0)] < \mathbb{E}[f(x)] \; \forall \, x \neq x_0 \in X$ as required.

Now without loss of generality let $f(x_0) = y_0$ and $A(\{(x_0, y_0)\}) = x_1$. Then the posterior, $P(f|\{(x_0, y_0)\})$ is again uniform where it is non-zero, as (pre-normalization) either $P(f|\{(x_0, y_0)\}) = 0$ or $P(f|\{(x_0, y_0)\}) = c$, where $c = \alpha$ if $y_0 < \bar{y}$ and $c = \beta$ otherwise. Note the likelihood $P(D|f)$ and thus the posterior can never be 0 everywhere as $\mathcal{F}$ contains all functions $f : X \to Y$.

Observe that redistributing the weights over the non-zero elements of the posterior without changing the total doesn't affect the previous behaviour as $f(x_0) = y_0$ for all non-zero weighted f in the posterior. Thus we now redistribute the weight over all non-zero weighted f in the posterior such that $\mathbb{E}[f(x_1)] < \mathbb{E}[f(x)] \; \forall x \neq x_1 \in X$. We can now repeat this process until every $x \in X$ has been sampled. As X is finite this process will eventually terminate.

□

3.6 Importance of the On-Policy Restriction

Theorem 23 above is stated and proved subject to the on-policy restriction, that is, the theorem is shown to be true only for the on-policy behaviour of the algorithm. In this section we show that this restriction is in fact necessary for the theorem to hold in general.

Let $X = \{0, 1, 2\}$ and $Y = \{0, 1\}$. Then we have $2^3 = 8$ possible functions $f : X \to Y$ which we can represent as the eight possible three bit binary strings, where the ith bit in the string is $f(i)$. Thus 010 represent the function where $f(0) = 0, f(1) = 1$ and $f(2) = 0$. A prior over Y^X consists of eight values, $p_1, \ldots, p_8$, giving the prior probabilities for each of the possible functions: $P(000) = p_1$, $P(001) = p_2$, ..., $P(111) = p_8$.

Suppose we have a sampling policy, A, with the following behaviour:

- $A(1??) = 1$ (that is, given that $f(0) = 1$, A samples at $x = 1$).
- $A(?1?) = 2$
- $A(??1) = 0$

It follows that if we wanted to construct a prior such that equation (3.4) produced

a sampling policy with the same behaviour as A, using (3.2) as the cost function, then we would have the following requirements on p_4, p_6, p_7:

$$A(1??) = 1 \Rightarrow p_6 > p_7$$

$$A(?1?) = 2 \Rightarrow p_7 > p_4$$

$$A(??1) = 0 \Rightarrow p_4 > p_6$$

Comparing the implications show us that no such prior exists, thus is not always possible to represent the behaviour of a sampling policy on all possible input data within the Bayesian framework described above. However, restriction to the on-policy behaviour is a natural constraint which avoids this issue. The problem of off policy behaviour is considered in more detail in Section 4.3.

3.7 Complexity Result

Using the mappings above we can now proceed with the second main result of the chapter. Let A be an arbitrary sampling policy. Then there is a prior P which produces the behaviour of A when combined with the greedy cost function, b, defined in (3.2).

Theorem 24.

$$K(P) \stackrel{+}{=} K(A)$$

Proof. P and b can produce A using a small program of complexity c_0 as in equation (3.4). Thus:

$$K(A) \leq K(P) + K(b) + c_0$$

However, $K(b)$ is also constant (i.e. independent of A and P). Thus we get:

$$K(A) \leq K(P) + c_1$$

Next we note that, from the proof of Theorem 23, A can produce the prior P via a simple recursive procedure, which can be performed by a small program with complexity c_2. Thus:

$$K(P) \leq K(A) + c_2$$

and thus:

$$K(P) \stackrel{+}{=} K(A)$$

□

Where $\stackrel{+}{=}$ means equal up to a constant. Note that as multiple priors can produce the same sampling policy via equation (3.4) we only differentiate priors up to the equivalence class induced by the surjection from priors to algorithm behaviours used in the proof above. The complexity of P is thus the complexity of the least complex member of the equivalence class to which it belongs. We will give a more detailed examination of the relationship between priors and behaviours in Section 4.2.

Note that the choice of the cost function b is flexible. All that is required is that $K(b)$ is constant (i.e. independent of A and P), and that for a given A, a prior P can be created for the chosen b using a fixed length program.

3.8 Simultaneous Interpretations

The above result was proved for the greedy cost function given in (3.2). However, it seems that it is extensible to a class of cost functions, and we now explore that possibility. Firstly and observation: It can be seen that there are cost functions for which no suitable prior can be constructed (most clearly, cost functions that make no use of the prior, for example $b(x,P) = x$). Thus, there is some non-trivial subset of cost functions for which Theorem 23 holds.

Define the set of *admissible cost functions* as those for which Theorem 23 holds. All optimisation algorithms are behaving Bayesian optimally with respect to all admissible cost functions simultaneously, each with respect to a particular prior.

Moreover, from Theorem 24 all fixed complexity cost functions are behaving with respect to priors of equal complexity, up to a constant. Thus in this regard there is no reason to choose one interpretation over another, they are all equally simple. We now explicitly illustrate this multiple interpretations principle.

So far we have looked exclusively at the greedy cost function. Another common cost function is expected information gain. Maximizing information gain can be expressed as minimising the expected entropy after sampling. Thus, the cost function to minimize is:

$$b_e(x,P) = \sum_{y \in Y} P(f(x) = y) H(P(\cdot \,|\{(x,y)\}))$$

Where $P(f(x) = y)$ is the probability $f(x) = y$, i.e. $\sum_{f \in \mathcal{F}} P(f)\delta(f(x) = y)$, and H is the entropy:

$$H(P) = -\sum_{f \in \mathcal{F}} P(f)\log(P(f))$$

Using b_e we can write down the information gain maximizing policy as:

$$B_e(D) = \operatorname*{argmin}_{x \in X} b_e(x, P(\cdot|D)) \tag{3.5}$$

In Figure 1 we show how both the naive greedy cost function B_g defined in 3.2 and the information gain maximizing policy B_e defined in 3.5 can be made to produce a required behaviour by selecting suitable priors.

Our investigations suggest that B_e is also an admissible cost function. The intuitive idea is to follow a level-by-level tree construction similar to that used in the proof of Theorem 23. Note that at each choice of x to sample, entropy minimization will select the x which is expected to yield the least uniform posterior, i.e. the posterior with the lowest entropy. Thus, each desired choice should be made to decrease the entropy by some fraction r of the amount the choice at the level before decreased the entropy by, with the other choices staying as uniform as possible. The fraction $r > 0$ must be sufficiently small so as to not cause choices on the lower levels to alter the choices already made at the higher levels, but r can be chosen to be arbitrarily small.

3.9 Caveats and Limitations

In this section we consider some possible problems of the approach to understanding optimisation that we have been exploring. The first potential issue is that probability distributions over functions are hard to represent, understand, interpret or visualise. It is true that probability distributions over function are hard to deal with in general, but those that correspond to optimisers are necessarily those that are more readily representable, in that they are represented already in some compact way in the optimiser itself. Moreover, in Section 3.7 we saw that the complexity of the optimiser implies a constraint on the complexity of the prior. A potential challenging project, in line with the efforts in [57], would be to develop mappings between natural language descriptions of problem function properties, e.g. “the functions are generally smooth, but have small scale noise” and suitable meta-heuristic optimisers. A mapping the other way would also allow an arbitrary optimiser’s behaviour to be broadly described.

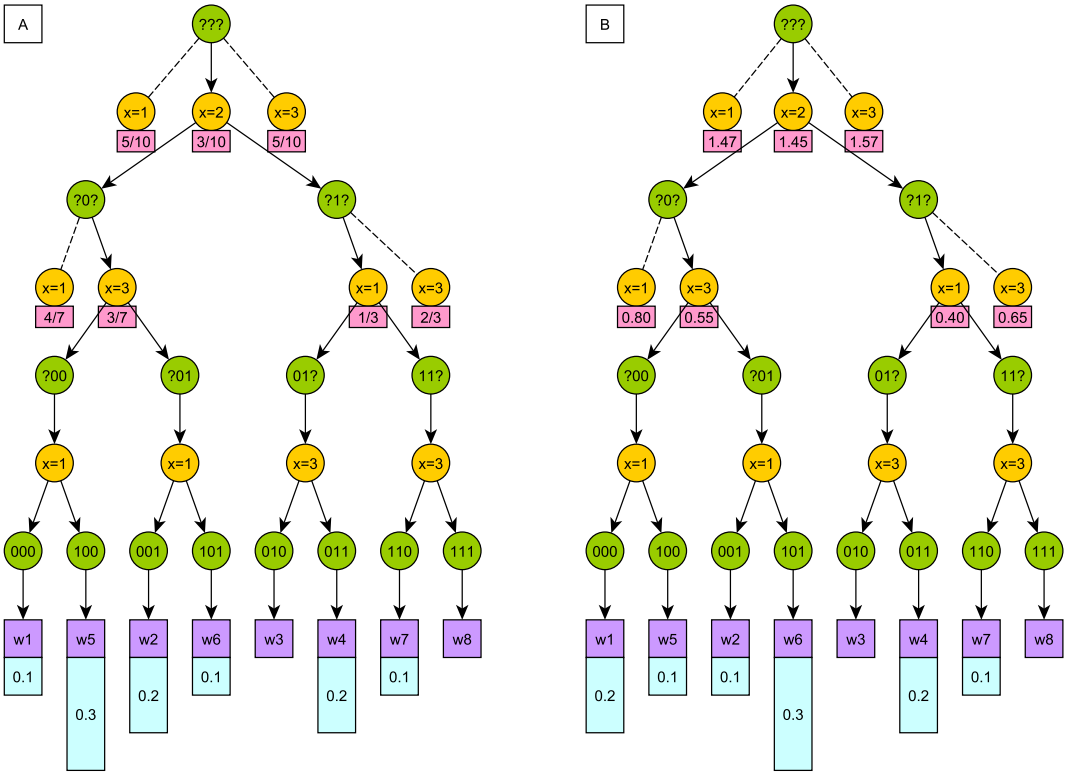

Figure 3.1: This figures shows a particular on-policy behaviour for two cost functions. **(A)** shows the behaviour of the greedy cost function B_g and **(B)** shows the behaviour of the maximum information gain cost function B_e. In this toy example $f : \{1,2,3\} \to \{0,1\}$, and a prior consists of a probability for each of the eight (2^3) possible functions. Suitable priors has been selected for each cost function so that their Bayesian optimal behaviours are identical. Green nodes show the current state of knowledge about f, and lead to various sampling choices shown as yellow nodes. The results of off-policy choices (dashed lines) aren't shown as they are never reached. In each tree the prior probability of each of the eight possible functions is shown at the associated leaf in a blue box. Larger boxes represent greater probability. The pink boxes show the cost assigned to each of the possible next sample points. Tree **(A)** shows the behaviour of the greedy cost function B_g and so the pink boxes show the expected $f(x)$. Tree **(B)** shows the behaviour of the maximum information gain cost function B_e so the pink boxes show the expected entropy of the posterior after sampling the corresponding x. The algorithm always samples at the x with the lowest cost. It can be seen that, as a result of the priors chosen, this produces the same behaviour in both trees.

A second difficulty is that we need to know the true distribution of problems we are dealing with in order to know the effectiveness of an algorithm, and thus to decide which optimiser to use. Although this is in some sense an issue, the fact is that it is not a binary situation; the more we know about the problems the better, and anything we know can in theory be used to better select the optimiser.

3.10 Discussion

We have introduced the set of admissible cost functions — cost functions that all sampling policies can be interpreted as behaving Bayesian optimally under with respect to some prior — and have shown that the greedy cost function is admissible. Thus, on-policy behaviour of any sampling policy can be interpreted as Bayesian optimal behaviour with respect to some prior under the greedy cost function. We have also argued that information gain is admissible, and that all sampling policies can be thus seen as simultaneously optimally greedy and optimally information gaining with respect to different priors.

Further, we have shown that the complexity of a sampling policy is equal to the complexity of the prior with respect to which it is optimally greedy, up to a constant factor. As a result the length in bits of an implementation of a sampling policy gives an upper bound on the complexity of the prior it implies. Additionally, it is observed that this generalizes to any small fixed complexity admissible cost function. A diagram capturing a high level view of the ideas presented in this chapter is given in Figure 3.2.

For a fully general perspective, further investigation should be made into algorithms with stochastic behaviour, algorithms where sampling is noisy and also the continuous case. The generalization of information gain to the continuous case with Gaussian noise has been carried out in [58], suggesting that such expansions are indeed possible. In the next chapter we will extend consideration to stochastic optimisers.

We have shown a relationship between the Kolmogorov complexity of priors and algorithms. It may be fruitful to investigate whether there are similar relationships between the time-complexity or space-complexity of priors and optimisation algorithms behaving with respect to these priors.

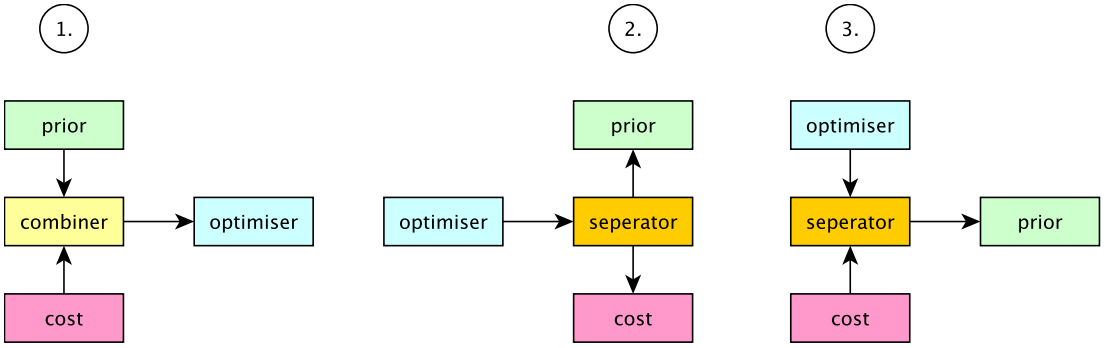

Figure 3.2: The figure shows: (1). A representation of optimisation algorithm construction in line with Equation 3.1. (2). A decomposition into prior and cost. Note that as discussed above, there are in fact may such possible separators performing different but valid compositions. (3). Given an admissible cost function a prior can be constructed.

Chapter 4

Broadening the Definition of Optimiser

4.1 Introduction

In the previous chapter we proved that the on-policy behaviour of any deterministic optimiser is the same as the on-policy behaviour of some Bayesian optimiser using the greedy cost function. In this chapter, among other things, we look at removing the on-policy and deterministic restrictions, such that the result holds for a broader set of optimisers.

To this end we examine stochastic optimisation (which we have so far discussed only briefly in the no free lunch setting), consider so-called forgetful algorithms, and look at meta-heuristics. However, before we do that we first examine some details of deterministic optimisers, and in particular, some of the less straightforward aspects of the Bayesian characterisation.

In Section 1.2 we commented on how optimisation problems could be difficult in various respects, and in this chapter we look at properties of optimisers that are used in real applications on these more difficult problems. In other words, we now expand our definition of optimiser in that it applies to real meta-heuristics.

The chapter proceeds as follows: first we will refine and clarify or current understanding of the Bayesian framework by looking at deterministic behaviours in more detail in Section 4.2, and then examining full policy behaviour in Section 4.3. Next we broaden our current definition of optimiser by introducing forgetful optimisers in Section 4.4 and stochastic optimisers in Section 4.5. Lastly, we look at how the Bayesian framework we have been developing can be used to examine the effects of common meta-heuristic strategies in Section 4.6.

4.2 Deterministic Optimisers

So far we have been mainly focused on understanding deterministic algorithms. In this section we continue towards this goal, discussing some more subtle details of the Bayesian approach in the previous chapter.

4.2.1 Mapping Behaviours to Priors

Previously we have seen that given an algorithm and an admissible cost function there exists a probability distribution over functions such that the algorithm's on-policy behaviour is just Bayesian optimal behaviour with respect to the cost function and the probability distribution. However, there are infinitely many possible probability distributions over functions, but only a finite number of different deterministic behaviours. As a result some behaviours must be produced by multiple priors, and so given a behaviour we can't necessarily produce a *unique* prior that produces it. In other words for a given behaviour and cost function admissible priors are not unique.

4.2.2 Behaviour Regions

In fact, under the greedy cost function each optimisation algorithm behaviour corresponds to a convex region in the space of priors. To see this it is enough to observe that a particular decision at a node corresponds to the prior satisfying a set of linear inequalities, and the feasible region defined by a set of linear inequalities is necessarily convex. Figure 4.1 aims to clarify this relationship between regions and behaviours, and Figure 4.2 shows some real behaviour regions.

A probability distribution over a finite set X where $|X| = n$ can be represented uniquely by a list of n positive numbers that sum to one, and every list of n positive numbers that sums to 1 represents a unique prior. Because of this bijection we can think of the space of priors as the positive part of the surface of an n dimensional sphere under the L^1 norm. Simply put, a prior can be represented as an n-element vector, where the elements sum to one, and none of the elements are negative. These vectors describe a surface, and behaviours correspond to reigons on this surface.

4.2.2.1 Random Priors

We performed an experiment where a prior was uniform randomly selected, and the behaviour it produced was calculated, we repeat this process and counted the number

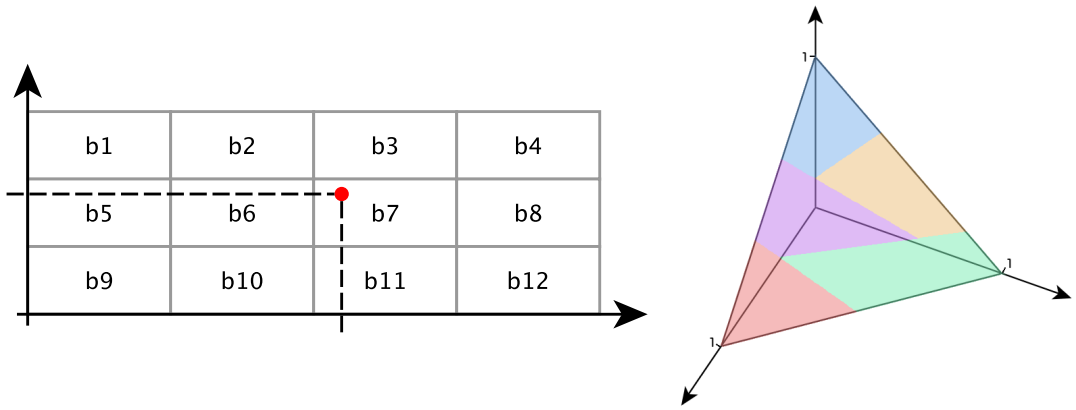

Figure 4.1: Left: A graphical representation of the space of possible priors showing the way that a particular optimisation behaviour corresponds to a region of the space of possible priors, and a prior distribution result in the behaviour of the region it falls in. In the diagram there are 12 behaviours, $b1, \ldots, b12$, and a prior represented by the red dot which would produce behaviour $b7$. As we will see this grid layout of behaviours in prior space is not actually realistic. Right: A more realistic graphical representation of the space of possible priors. The positive "quadrant" of the surface of the sphere (under the L^1 norm) is the space of possible priors, and the coloured regions show the corresponding behaviours

of times each behaviour occurred. The result should be that the proportion of times a behaviour occurs is the same as the fraction of the surface its region covers. The results are shown in Figure 4.3 and detailed in Figure 4.4. As these figures show, uniform randomly selecting a prior is not the same as uniform randomly selecting a behaviour. This result is interesting because it shows that some behaviour regions are larger than others. It also means that if we assume that all probability distributions over functions are equally likely to describe the problems we intend to optimise, then some optimisers are better than others. This is a novel free lunch result.

4.2.3 Cost Functions

In this thesis we have only really considered two cost functions, greedy and information gain maximising, and in fact information gain maximisation has only been discussed briefly. It seems reasonable to wonder what we can we say about cost functions in general. Are some cost functions better than others in some sense? For example, do perhaps some cost functions lead to more interpretable priors for popular optimisers? Are some cost functions able to characterise more full behaviours than others? Are any

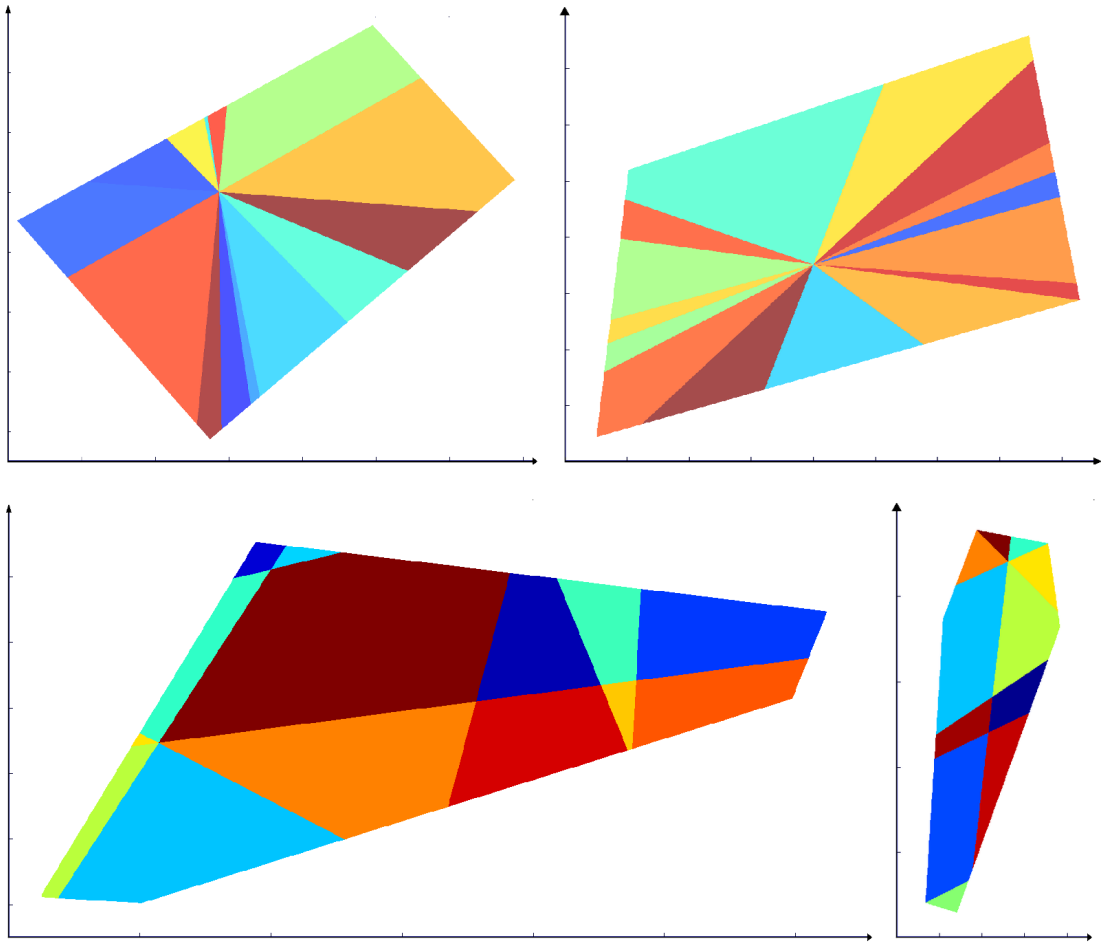

Figure 4.2: Each of the graphs shows a two-dimensional cross-section of the space possible probability distributions over functions mapping from $X = \{1,2,3\}$ to $Y = \{0,1\}$. As there are $|Y|^{|X|} = 3^2 = 8$ possible functions this is just a two-dimensional cross-section of $[0,1]^8 \in \mathbb{R}^8$. Each point on the graph is coloured according to the behaviour it produces under the greedy cost function. Note that these colours are random and similar colours do not imply similar behaviours. White indicates that a point isn't a valid prior and so has no resulting behaviour. These non-valid priors occur when the probabilities do not sum to one, or aren't all positive. Note that, as discussed in the main text, behaviour lie in convex regions. The top two cross-sections are centred on the uniform distribution, which is the point at the centre from which all the behaviours fan out. As can be seen very many behaviours are accessible by slight deviations from the uniform distribution. The bottom two figures are arbitrarily selected cross-sections. The bottom left cross-section is examined in detail in Figures 4.5 and 4.5 below.

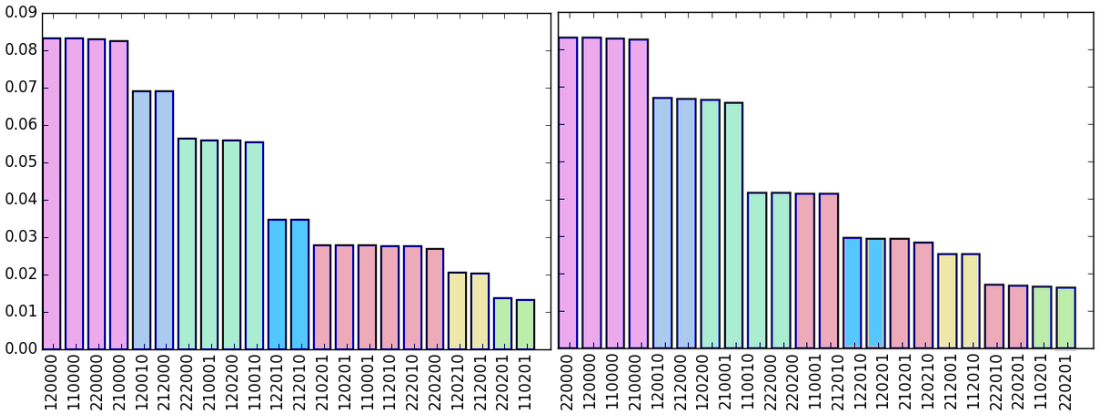

Figure 4.3: Bar graphs showing the probabilities of the 22 full optimiser behaviours, for optimisers on functions mapping $\{0,1,2\}$ to $\{0,1\}$ that initially sample at $f(0)$, that can be achieved under the greedy cost function from a single prior. The left graph shows the optimisers probabilities when the probability distributions over functions are uniformly sampled. The right graph shows the resulting probabilities when the probability distributions are constructed by uniform randomly selecting a probability for each function, then normalizing such that they all sum to one. Note that there are 64 full behaviours that initially sample at $f(0)$, and 42 of them do not result from any prior under the greedy cost function. On the x axis, the labels consist of 6 numbers which correspond to the optimisers sampling decisions (i.e. whether to sample $f(0)$, $f(1)$ or $f(2)$ next) for the data $0??, 1??, ?0?, ?1?, ??0$ and $??1$ in that order.

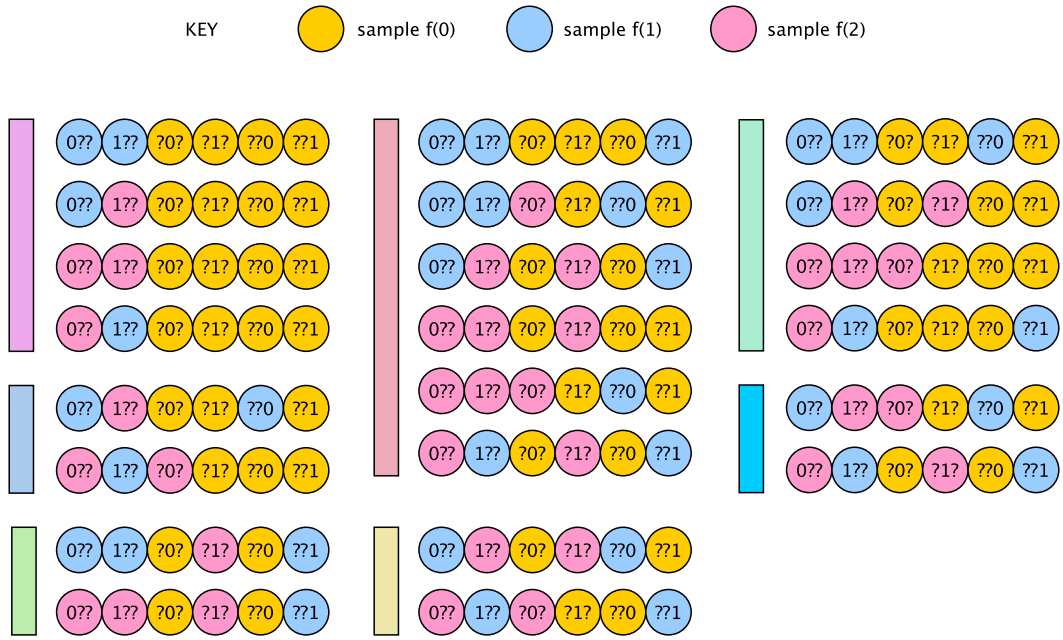

Figure 4.4: The 22 possible full deterministic behaviours that initially sample $f(1)$ under a greedy cost function. Here we also use a succinct color-key representation of the behaviour trees, so that a behaviour can be conveniently represented on a single line. The blocks on the left of the behaviours show the colour of their corresponding bars in Figure 4.3.

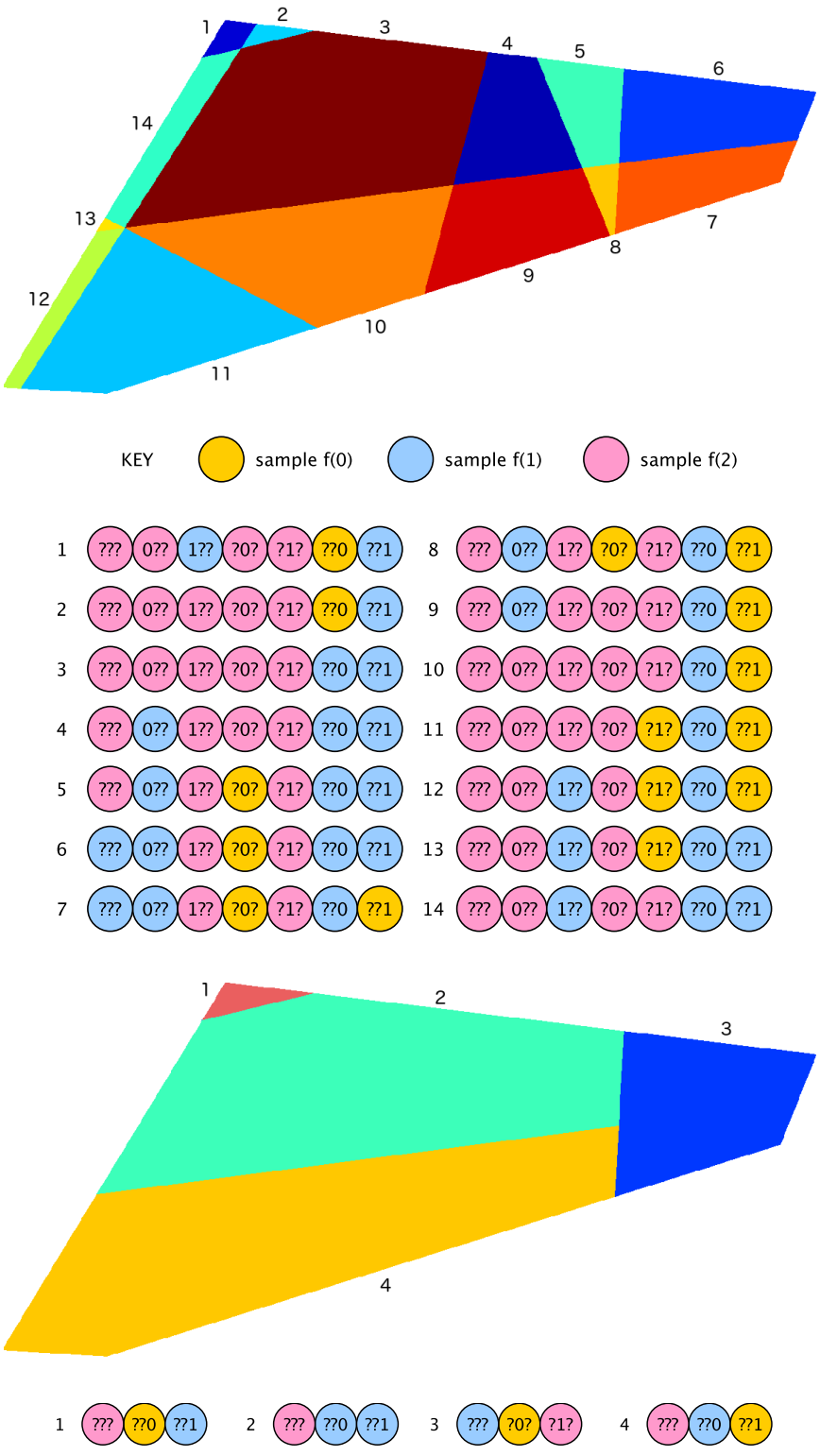

Figure 4.5: Top: A two dimensional cross section of the space of possible priors over functions mapping from $X = \{1,2,3\}$ to $Y = \{0,1\}$ with the distinct full behaviour regions coloured and labelled. It can be seen that behaviour regions sharing an edge differ in their behaviour at a single node. The edges are the linear boundaries between behaviours. Bottom: A two dimensional cross section of the prior space with the distinct on-policy behaviour regions coloured and labelled.

commonly used cost function not admissible? What are the necessary and sufficient conditions for a cost function being admissible? These questions are outside of the scope of this thesis however, and we will for the time being have to proceed knowing only that admissible cost functions do exist, and that the two we have considered are examples of such.

4.2.4 Measuring Behaviour Performance

Roughly the claim is that a behaviour does well on the probability distribution that it implies, however as we have seen a behaviour could actually be the result of a set of probability distributions. Let A be an algorithm carrying out a deterministic behaviour b and let P_b be the set of all priors that result in that behaviour under the greedy cost function. How does A's behaviour vary over elements of P_b? Does the algorithm do better on some priors in this set than others? As the example below shows, the prior that induces a behaviour can result in various performances. Essentially we know that the optimiser is taking the best decision, but that doesn't tell us how good the best decision is in an absolute sense.

Example 13. *Consider an optimiser A for functions mapping between* $X = \{1,2\}$ *and* $Y = \{0,1\}$. *Assume the A always samples* $f(1)$ *first, i.e. that* $A(??) = 1$. *Let us assume also that A is greedily minimising, and that the cost of the sample is just its corresponding Y value. Clearly* $A(??) = 1$ *is the Bayesian optimal strategy if and only if* $\mathbb{E}[f(1)] \leq \mathbb{E}[f(2)]$, *but this doesn't actually determine the performance of the optimiser.*

Thus, a better way of understanding an implied behaviour is that it is the behaviour that does as well as possible, although in absolute terms this might not be well. Also, knowing the behaviour doesn't tell you the expected performance. It tells you what is best to do, but not how good that best approach is.

4.2.5 Deterministic Optimiser Summary

In this section we have looked at the Bayesian framing of deterministic optimiser in more detail. We have seen that a single behaviour corresponds to various priors, and under the greedy cost function the priors for a given behaviour form a connected convex region of prior space. We have also seen that randomly sampling a prior is not the same as randomly sampling a behaviour, which is a free lunch result of a sort not

addressed by the no free lunch results. Further, we have seen that there are a range of performances possible for a given behaviour.

There are many other potential lines of investigation for further understanding deterministic optimisers. We could investigate the connectivity of regions, exploring how many behaviours neighbour a particular behaviour for example.

4.3 Full Behaviours

Previously we have seen that for deterministic algorithms, given a cost function, the on-policy behaviour always implies a prior over functions. However, we have also seen that when we expand consideration to full behaviour, and consider not just the on-policy behaviour but also the off-policy behaviour, then we encounter some difficulties when we try to infer prior beliefs from the behaviour.

Essentially, the problem is that some full behaviours are seemingly irrational. That is, there is no probability distribution over functions that results in their behaviour with respect to the greedy cost function we have so far been using. We briefly examined this problem in Section 3.6. Below we work through a more detailed example of the problem.

4.3.1 The Full Behaviour Problem

Let A be an optimisation algorithm for functions $f : \{0,1,2\} \rightarrow \{0,1\}$. We want to try to understand the behaviour of A in terms of beliefs about the problems it will encounter. We can decide on our cost function, here we will assume that it is greedily searching for minima, so that at each point it selects the x value to sample that has the lowest expected $f(x)$. The cost function, then, is:

$$c(x,d) = \mathbb{E}[\, f(x) \mid d \,] \text{ (cost function)}$$

Given the cost function, a particular decision made by the optimisation algorithm corresponds to a belief about the probabilities of functions. For example, if the optimiser initially samples at $f(0)$, then we can infer certain inequality relationships. For example, $A(???) = 0 \implies \mathbb{E}[f(0)] < \mathbb{E}[f(1)]$. This in turn implies that $P(f(0) = 1) < P(f(1) = 1)$, and thus $P(100)+P(101)+P(110)+P(111) < P(010)+P(011)+P(110)+P(111)$ which simplifies to $P(100)+P(101) < P(010)+P(011)$. $A(???) = 0$

also implies that $\mathbb{E}[f(0)] < \mathbb{E}[f(2)]$, and further inequalities on the prior probabilities can be deduced from this in the same way.

In this way, the behaviour of an algorithm implies various constraints on the prior beliefs. We have seen that when considering only the on-policy behaviour these constraints are never unjustifiable. In the case of full behaviour however, sometimes the implied beliefs are inconsistent. An example of this problem is given in Figure 4.6.

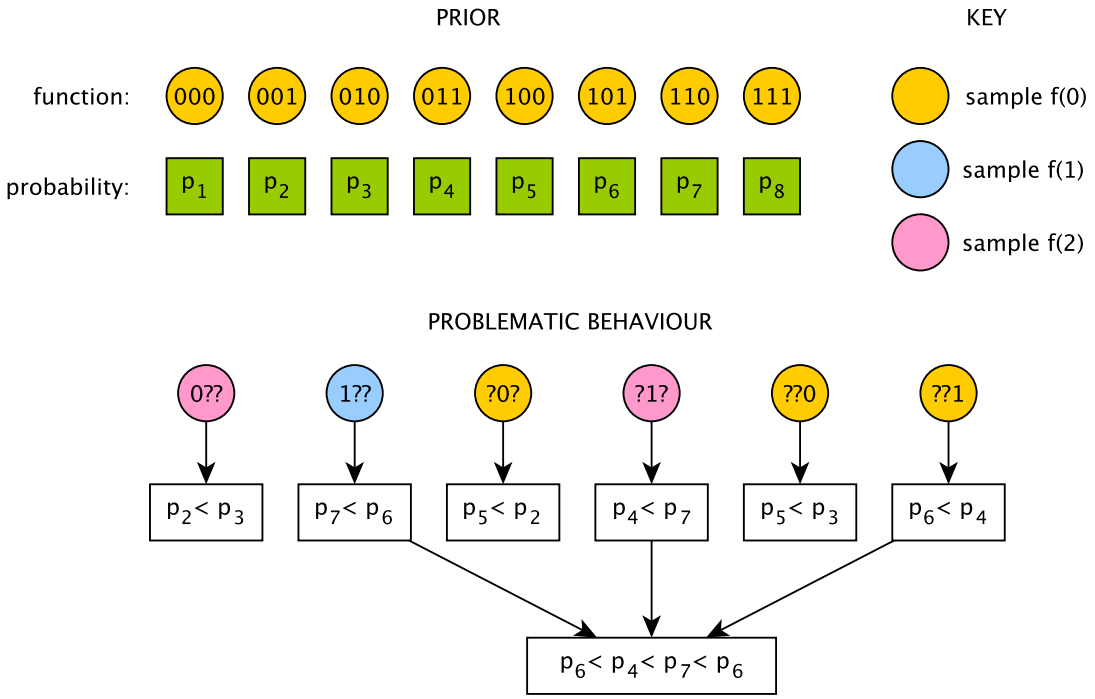

Figure 4.6: Example of a problematic full behaviour. The white boxes show the inequalities between the probabilities of the functions implied by the decision at the node they are attached to. For example, at node $1??$ the next sample is at $x = 1$ yielding either $10?$ or $11?$. As a result the algorithm must believe that, given $f(0) = 1$, then $P(f(2)) = 0) < P(f(1) = 0)$. That is, $P(100) + P(110) < P(100) + P(101)$, implying that $P(110) < P(101)$, i.e. $p_7 < p_6$.

4.3.2 Resolving the Full Behaviour Problem

We have now seen how the full behaviour dilemma arises. In this section we consider what it means, in what sense it is a problem, and how it might be "resolved".

4.3.2.1 Restrict Attention to Consistent Optimisers

One important point is that the full behaviour problem is not an issue present in all optimisers. Some optimisers have a consistent prior under the greedy cost function for their full behaviour. For example, all the behaviours shown in Figure 4.4 are free from this problem.

When an optimiser doesn't suffer from the full behaviour problem it means that a single prior explains all of its behaviour. Everything the optimiser does, regardless of the data it is presented with, is consistent with a single coherent belief about the world. On the other hand, optimisers that suffer from the full behaviour problem behave in a way that seemingly can't be rational (or at least, not straightforwardly, but see Section 4.3.2.3 below for an alternative approach). It seems perfectly reasonable to only be interested in rational optimisers. Thus, one sensible move might be to restrict attention to only those optimisers that have consistent full behaviours. It is also worth noting that this in no way restricts the richness of on-policy behaviours, rather it simple forces the off-policy behaviour to *line up* with the on-policy behaviour in a natural way.

4.3.2.2 Ties and Tie-Breaking

Another potential resolution of the apparent inconsistency in the implied prior is to allow draws in the prior, and then have a tiebreaker mechanism. For example, in the example optimiser detailed in Figure 4.6 the problem is that the behaviour seems to require that $p_6 < p_4 < p_7 < p_6$, which is impossible. However, if we instead assume that $p_6 = p_4 = p_7 = p_6$ then this is consistent. If we are now given $f(0) = 1$, then $P(f(1) = 0) = P(f(2) = 0)$, and so we need a tie-breaker rule to decide where we should sample.

The problem with this approach however is that these tie-breaking rules may need to be arbitrarily complex, and essentially might amount to just listing the sampling decisions, which moves the whole understanding away from the natural Bayesian behaviour back toward arbitrary rule based behaviour with no clear underlying motivation.

4.3.2.3 Conditional Priors

We now consider a viewpoint from which the behaviour of all the optimisers remains Bayesian. A potential interpretation which keeps all the full behaviours Bayesian is to allow the prior to be conditional on the starting data. This would mean that the

algorithm has different beliefs about what functions are likely, depending on what data it is presented with. This interpretation is in some sense *scientific*. The idea is that the data we are presented with is telling of the underlying structure in some way over and above just the literal content of the data.

4.3.3 Full Behaviour Summary

In this section we re-examined full policy behaviour, and in particular, we looked more closely at the existence of inconsistent full behaviours. We have seen that restricting attention to only coherent full-policy algorithms is one option, which doesn't alter the possible on-policy behaviours, but means all optimisers are consistent. However, some popular optimisers may well turn out to not be consistent in their full behaviour, in which case it may be necessary to focus only on the on-policy behaviour. This is not an issue theoretically, as essentially all optimisation research restricts attention to the on-policy behaviour.

4.4 Forgetful Optimisers

So far we have only considered optimisation algorithms that remember every point they sample over the entire course of the optimisation. In this setting, as the algorithm explores a function it builds a history of visited x values and their corresponding y values, and when determining where to sample next the decision can potentially depend on all those previous results. Some real optimisation algorithms do keep track of all the points visited and use them to inform their future sampling decisions, but there are also algorithms that do not. Some algorithms, for example, forget all but a selection of the visited points or store some lossy statistics of the samples. They then use this lossy memory of the past to direct their future behaviour.

In this section we expand our concept of optimisation algorithms to include those in this latter category. We will call algorithms that do something other than simply remember everything they have seen *forgetful*, and so this section is dedicated to considering *forgetful algorithms*. We will still restrict our attention to deterministic algorithms.

As we saw in the previous chapter, all non-forgetful optimisers can be understood as behaving optimally with respect to a prior and cost function. Can forgetful algorithms also be understood in this Bayesian way? If so, we can ask various pertinent

questions, but perhaps most importantly, we can ask how different strategies for compressing previous samples affect the algorithm's implied prior.

When is it good to keep track of sampling histories in order to use them to guide future sampling? When is it better to forget? The general idea is that when function evaluations are expensive, we try to build better surrogate models and make use of those to avoid sampling the real function too much. On the other hand, when functions are easy and cheap to evaluate we are often better off using our time and computational resources actively evaluating the real function. This intuition makes sense, but can it be more rigorous?

Maximally exploiting the past samples in practice means conditioning the prior on *all* points seen so far, yielding the full posterior. This is often expensive or even impossible. In the case of a Gaussian process prior the complexity grows cubicly in the number of points sampled. More specifically, a high end personal computer as of 2015 can invert $n \times n$ matrices for n up to a few tens of thousand in a few minutes. This becomes quickly prohibitively expensive in real optimisation tasks.

As a result of this problem, many techniques have been developed to try to alleviate this computational cost issue. A common approach is to try to find a small number of data points m that in some way *summarise* the large number of real data points n. If this can be done then you reduce the complexity from n^2 to m^2 with $m \ll n$ [59].

Let $s(d) = x$ be the actual application of the sampling policy to the known data. This has some cost related to the input d, for example in the Gaussian process case larger data sets have larger cost, although different sampling policies have complexity related to the inputs to greater or lesser degrees. For example, the time taken by the random search sampling policy is independent of the current knowledge of the problem.

In order to reduce the complexity of computing the next sample we use some summarised data $\hat{d} = g(d)$. In fact this produces a new optimisation algorithm $\hat{s} = s \circ g$, $\hat{s} : \mathcal{D} \to X$. It is thus the case that summarising the data is equivalent to adjusting the optimisers prior. If summarising the data allows the sampling policy to be computed more efficiently, then the summarising has necessarily resulted in a more easily computable prior. Work on how various data summary techniques affect a Gaussian process optimiser can be found in [60]. The conclusion is that essentially the different summary techniques just alter the Gaussian process's prior in some way.

Here we show that this idea generalises to all optimisers. Any optimiser with a scheme for summarising the history can instead be seen as an optimiser with no data

summary scheme, but a simpler prior.

4.4.1 Definitions

We start with a preliminary definition, then give a formal definition of forgetful sampling policies and forgetful algorithms.

Definition 19 (Iterative Compression Strategy)**.** *Let X and Y be finite sets, $\mathcal{F}$ be the set of all possible functions from X to Y, $\mathcal{D}$ be the set of all possible data for functions in $\mathcal{F}$, and C be an arbitrary finite set.*

We call $\alpha : \mathcal{D} \to C$ an iterative compression strategy iff there exists a function $g : C \times X \times Y \to C$ such that for all $d \in \mathcal{D}$, $x \in X$ and $y \in Y$, $g(\alpha(d), x, y) = c \implies \alpha(d|(x,y)) = c$ (where $d|(x,y)$ is the set of samples in d plus sample point (x,y)).

The arbitrary finite set C in the above definition can be thought of as the set of possible compressed or lossy versions of the sample history, and $\alpha : \mathcal{D} \to C$ is then the compression operation, or compressor. For example, α might forget all the previous sample details except for the position and value of the maximum. In this case $C = X \times Y$ and, for example, $\alpha(\{(0,3),(2,7),(8,5)\}) = (2,7)$.

The restrictions on the compressor α ensure that the algorithm *only uses the compressed version of the previous data*, along with the new sample result, to construct the new compressed data. This allow the algorithm to avoid requiring an internal state on which the behaviour is conditioned. If it wasn't for this restriction, then there might exist $d_1, d_2 \in \mathcal{D}$, $x \in X$ and $y \in Y$ s.t. $\alpha(d_1) = \alpha(d_2)$ but $\alpha(d_1|(x,y)) \neq \alpha(d_2|(x,y))$. This would be problematic as, if d_1 and d_2 both result in the same compressed state then they should produce the same new compressed state when they each have the same new data point appended. We require the next compressed state to depend only on the current compressed state and the latest data point.

It is worth noting that the definition of compressor we present is order invariant in the sense that the compressed representation is required to depend only on the set of (x,y) pairs that have been sampled by the algorithm and not the order in which the samples were made.

An alternative representation would be an algorithm that kept some internal state, and then mapped $S \times X \times Y \to X$, where S was the set of potential internal states. Represented this way the algorithm would be a function, conditioned on some internal state, mapping the latest sample point (x,y) to a new point to sample in X. The internal state would serve as the compressed memory of all the previous sample results.

However, the representations are equivalent, and the one we have chosen avoids internal states, makes the compression of previous data an explicit function, and allows the algorithm to be defined as mapping $\mathcal{D} \to X$ (rather than $X \times Y \to X$), which is in line with or definition of non-forgetful optimisers. We now use the concept of an iterative compression strategy to define forgetful sampling policies and algorithms.

Definition 20 (Forgetful Sampling Policy)**.** *Let X and Y be finite sets, f be an arbitrary function $f : X \to Y$, $\mathcal{F}$ be the set of all such f, $\mathcal{D}$ be the set of all possible data for functions in $\mathcal{F}$, C be some finite set, and $\alpha : \mathcal{D} \to C$ be an iterative compression strategy.*

Let $s : \mathcal{D} \to X$ be defined as $s(d) = a(\alpha(d))$, where $a : C \to X$ is an arbitrary function. We call such an s a forgetful sampling policy.

Here the function $a : C \to X$ determines where next to sample, based on compressed version of the sample history, $c = \alpha(d)$. Thus, this decision is made without access to the full data d.

Definition 21 (Forgetful Optimisation Algorithm)**.** *A forgetful optimisation algorithm A is iterative use of a forgetful sampling policy s.*

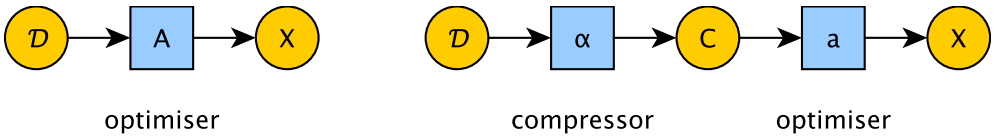

Figure 4.7: On the left a representation of a non-forgetful optimiser, which is given for comparison with the representation of a forgetful optimiser shown on the right. As can be seen, the forgetful optimiser has an additional compressor function, α, which transforms the input to the optimiser.

Figure 4.7 compares the composition of forgetful and non-forgetful optimisers. Having given a definition we can now consider the properties of forgetful optimisers, and contrast them with our original non-forgetful optimisers.

Firstly, forgetful optimisers contain non-forgetful optimisers as a sub-set. To see this we can set $C = \mathcal{D}$ and let α be the identity map. Also, it can be observed that the above definition allows for infinite looping behaviour. As a trivial example, if $|C| = 1$, e.g. $C = \{c\}$, then the algorithm will just repeatedly sample $x = a(c)$ (with $y = f(x)$ always resulting) and will never make any further progress in exploring f. This could

be thought of as being too forgetful, the algorithm does something, then immediately forgets and does it again, and so on.

We will restrict our attention to those forgetful algorithms that eventually explore the whole function, regardless of the function being optimised. We call these algorithms "eventually exploring forgetful optimisers" (EEFO).

As X, Y and C are finite, the necessary and sufficient conditions for a forgetful algorithm to be EEFO is that the algorithm does not get stuck in a loop before it has sampled the function at all $x \in X$. If we exclude the possibility of forgetful algorithms getting stuck in a loop, then forgetful algorithms are just resampling algorithms under a different name.

4.4.2 Graph Representation

Just as a non-forgetful optimiser can be represented as a behaviour graph with data as nodes, and edges representing the possible transitions resulting from sampling (see Definition 10), so forgetful optimisers can be represented as graphs, but now with nodes representing the compressed versions of the data.

Definition 22 (EEFO Behaviour Graph)**.** *The behaviour of an everywhere exploring forgetful optimiser can be represented as a directed graph in which the nodes represent the compressed states $c \in C$ from the optimiser's forgetful sampling policy, and the edges represent possible transitions resulting from sampling in accordance with the sampling policy. See the left hand graph in Figure 4.9 for an example of an EEFO behaviour graph.*

Theorem 25 (EEFO Behaviour Graphs can be Mapped to Trees)**.** *An eventually exploring forgetful optimiser's behaviour graph is acyclic at-least until every $x \in X$ has been sampled. If we stop the optimiser when every $x \in X$ has been sampled then all eventually exploring forgetful optimiser's behaviour graphs are equivalent to behaviour trees.*

Proof. From the fact that EEFO does not get stuck in a loop we know that its graph representation is loop free. Thus, the theorem is just a direct result of the fact that any finite graph without loops can be "unfolded" into a tree.

□

Figure 4.8 shows how an EEFO truncated behaviour graph can be unfolded into a tree. Figure 4.9 then shows how meaningful labels can be inferred for the compressed

states $c \in C$.

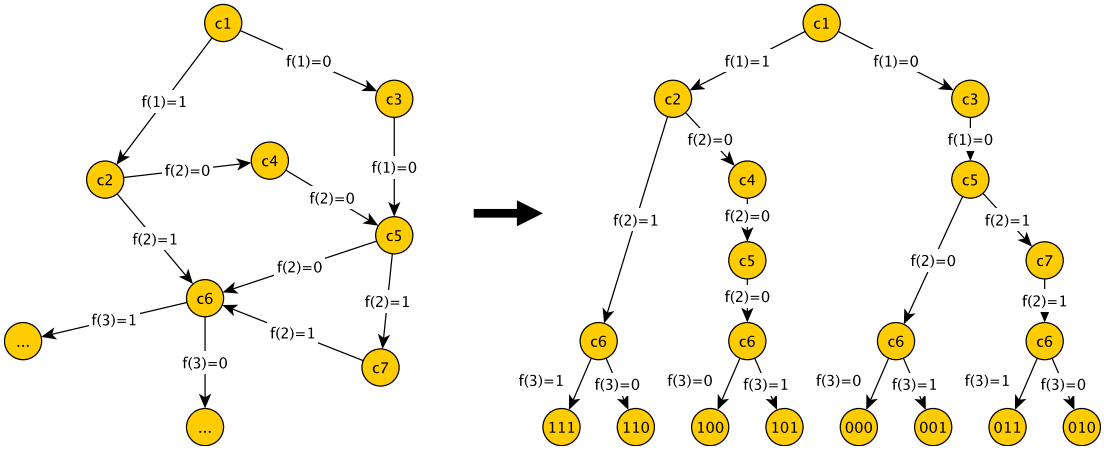

Figure 4.8: Unfolding the compressed behaviour graph. On the left is the behaviour graph showing the behaviour on the

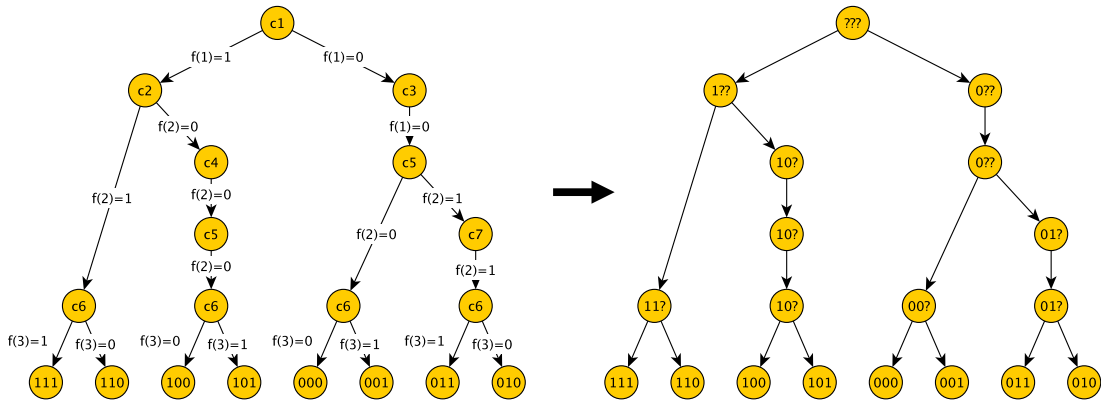

Figure 4.9: The behaviour of an example EEFO is shown on the left. As we have shown above, the graph is necessarily acyclic until all $x \in X$ have been sampled. For any EEFO graph, the state of knowledge (of an observer who doesn't forget) can be inferred for each node. The graph with the resulting inferred node labels are shown on the right.

We can now begin to prove other requirements that an EEFO must necessarily meet. We start by showing that in order to be eventually exploring, a forgetful optimiser must have its set of compressed sample data C be such that $|C| \geq |X|$.

Theorem 26 (Minimum Number of States)**.** *In order to be able to fully explore functions $f : X \to Y$, a forgetful optimiser must have $c \in C$ such that $a(c) = x$ for all $x \in X$. It follows that in order for a forgetful optimiser to be eventually exploring, $|C| \geq |X|$.*

Proof. The function $a : C \to X$ determines where the algorithm will next sample given compressed sample history c. If there exists some x such that there is no $c \in C$ for which $a(c) = x$ then x will never be sampled. As each x must be sampled to fully explore f then there must be at least as many states as there are x values, thus $|C| \geq |X|$. □

4.4.3 Ignoring Resampling

We saw in previous chapters, the (on-policy) behaviour of a non-forgetful optimisation algorithm can be represented as a tree. We have now seen that this is also true for EEFO. From 4.4.2 we know that an EEFO can't revisit any $c \in C$ until it has sampled every $x \in X$, and the behaviour of the algorithm up until this point can be represented as a tree. We now show that EEFO algorithms naturally surject onto non-forgetful optimisers. In fact, if we take a EEFO and simply ignore any resampling behaviour, then we obtain a standard non-forgetful optimiser. As an example the EEFO from Figure 4.9 is shown alongside its equivalent non-forgetful optimiser in Figure 4.10.

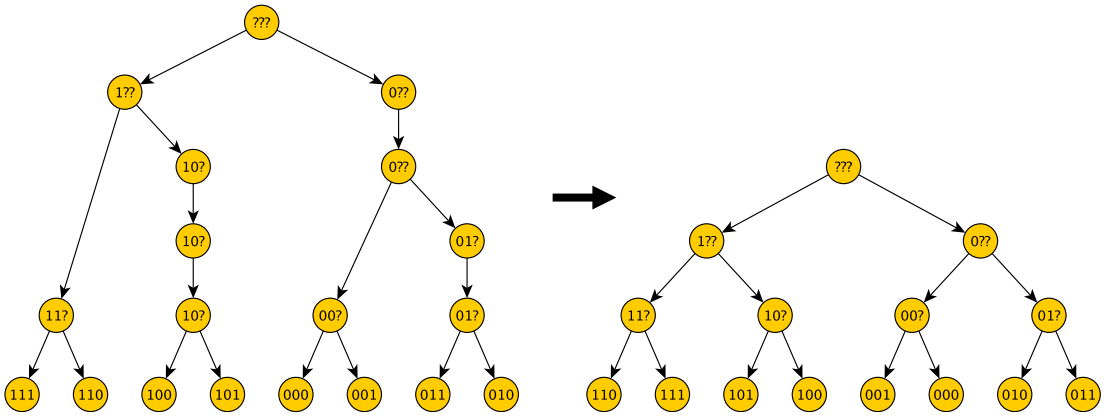

Figure 4.10: On the left is the behaviour graph for a EEFO. On the right the same behaviour graph is shown, with the resampling decisions skipped.

Thus by the three step process of unfolding the graph, inferring the labels of the nodes, and ignoring resampling we can see that the on-policy behaviour of any EEFO is equivalent to the behaviour of some non-forgetful optimiser, and thus can be interpreted as Bayesian optimal behaviour.

However, we have not really dealt with *why* the algorithm forgets and has to resample. If we are satisfied with considering resampling as meaningless behavioural blips that do not say anything about the EEFO beliefs then we can stop, and interpret the EEFO's prior to be the same as it's corresponding non-forgetful optimiser's. As we

have seen, forgetting does nothing more than introduce resampling. It does not allow any new behaviours. Thus, it seems that compression strategies may have computational benefits, but do not have behavioural ones.

4.4.4 The Effects of Forgetting

We now seek to understand how the forgetful nature of an algorithm can affect the prior distribution implied by the algorithm's behaviour. If an algorithm has only a lossy representation of the results of previous samples, as is necessarily the case if $|C| < |\mathcal{D}|$, then it cannot distinguish all possible situations, and as a result its behaviour cannot be as fine grained as a non-forgetful alternative. At the same time, as discussed above, allowing an optimiser to be forgetful does not result in a broader range of possible behaviours; any forgetful optimiser can be mimicked exactly by a non-forgetful one.

Forgetful optimisers don't allow new behaviours, but forgetfulness may well bias the sorts of behaviours possible towards particularly effective implied priors. Thus, although forgetful optimisers are restricted to a set of behaviours it may be that this restricted set is somewhat well aligned with common problems. The idea is that not being able to keep hold of too much information restricts your behaviour options in a way that tends to make them better.

4.4.5 Forgetful Optimisation Summary

In this section we have expanded our definition of optimisation algorithm to include those optimisers that store only some information about the previous samples, be that through simply only keeping some sample results, or by storing the previous results in some lossy compressed way. Many real world optimisation algorithms fall into the class of forgetful optimisers, particle swarm optimisation, gradient descent optimisers, and genetic algorithms to name but a few prominent examples (note however that only the second of these is generally deterministic). We then showed that on this extended class of algorithms it is still the case that all algorithms are describable as a prior and cost function pair. Lastly we considered how, although not capable of producing new behaviours, forgetfulness might favourably skew the sorts of behaviours produced by optimisers.

Forgetful optimisers, due to their forgetting, can have low memory requirements. However, it is worth observing that an ideal forgetful algorithm, i.e. one that samples without repetitions, can be realised only through systematic exploration behaviour.

This may well result in greater temporal complexity.

In summary, it is true that storing all the samples is necessarily maximally informative. If an algorithm stores everything it can always behave as if it has only stored some results later, if required. For this reason forgetfulness isn't beneficial behaviourally. Rather, it seems to have two possible non-behavioural benefits: it is often chosen for its computation efficiency and logistical benefits, and moreover, it may well coincidently produces good sampling behaviour.

4.5 Stochastic Optimisers

So far in this thesis we have almost exclusively focused on deterministic optimisation. However, in reality very many optimisation algorithms, quite possibly the majority, are not deterministic. Genetic algorithms, particle swarms, simulated annealing, random search and stochastic gradient descent, to name but a few, all use stochasticity in their optimisation behaviour.

Thus, in this section we shift our focus on to stochastic optimisation algorithms. Stochastic optimisers are a more general class of algorithms as they contain deterministic optimisation algorithms as a proper subset. As a result, in shifting our attention to stochastic algorithms we are actually just broadening our considerations.

In previous chapters we have presented a framework for understanding all deterministic optimisation algorithms as Bayesian. Here we will investigate whether we can extend this framework to include stochastic optimisers. As we will see, the answer is essentially yes, but there are some technical details that need to be carefully worked through in order to reach this result.

Before we proceed, it is worth highlighting the relationship between stochastic behaviour and the "everywhere exploring" optimisers discussed in Section 4.4. Practically speaking, stochasticity is a behaviour that, if correctly used, can be shown to mathematically guarantee that an optimiser will eventually sample everywhere in X.

Practically, however, interesting functions are usually more complex than available algorithms, such that stochasticity is a mathematical assumption which guarantees that the algorithm goes everywhere in X. E.g. a forgetful optimiser would not get stuck if it was stochastic, so stochasticity is a way to implement an exploring forgetful optimiser: Of course, it will resample a lot.

4.5.1 Defining Stochastic Optimisers

Informally, a stochastic optimisation algorithm is an optimisation algorithm that can involve randomness in its decisions. Whereas a deterministic optimisation algorithm always makes the same decision for a given input, a stochastic optimiser selects randomly from possible decisions according to a probability distribution. As a result, if given the same function to optimise multiple times, the stochastic optimiser may well make different sampling decisions on each occasion. Below we make this definition more rigorous. Additionally, Figure 4.11 shows diagrammatically an example of a stochastic optimiser's behaviour policy, which can be helpful for guiding intuitions. We first give two equivalent definition of a stochastic sampling policy, and then a definition of a stochastic optimisation algorithm.

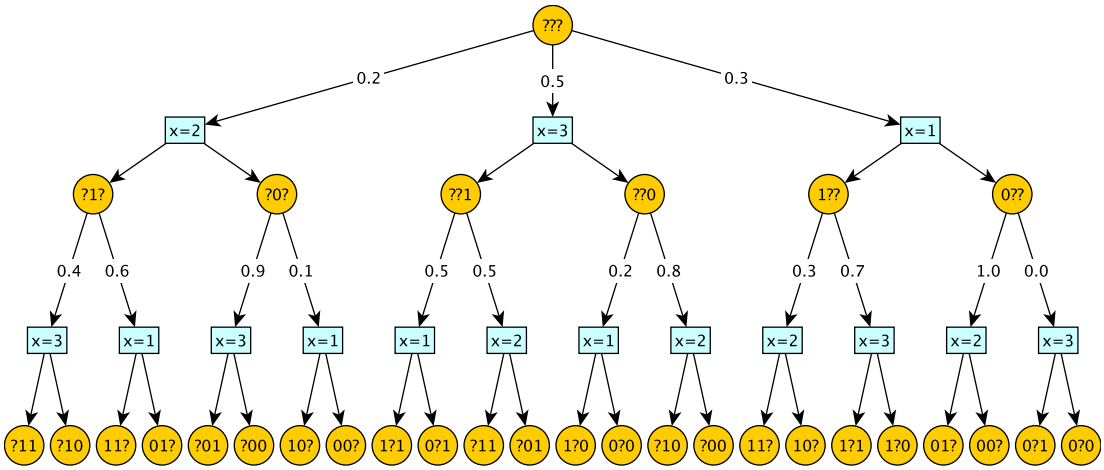

Figure 4.11: Example of a stochastic optimiser. Note that although the above diagram shows a tree, in reality the paths converge on the eight possible three-bit functions in the final layer, which isn't shown here for clarity. As discussed in the main text, the fact that the behaviour doesnt form a tree means that extra care needs to be taken when thinking about the Bayesian interpretation of stochastic algorithms.

Definition 23 (Stochastic Sampling Policy v1)**.** *Define $\mathcal{P}_X$ as the set of probability distributions over X. A stochastic sampling policy is a function $s : \mathcal{D} \to \mathcal{P}_X$.*

Definition 24 (Stochastic Sampling Policy v2)**.** *A stochastic sampling policy is a probability distributions over X parametrised by d, $P_X(\,\cdot\,|d)$.*

Definition 25 (Stochastic Optimisation Algorithm)**.** *A stochastic optimisation algorithm repeatedly uses a stochastic sampling policy to obtain a probability distribution*

over X, it then draws a sample from this distribution to determine which $f(x)$ to evaluate next, adding the results at each step to the data.

4.5.2 Comparing Stochastic and Deterministic Optimisers

We have now seen a definition of stochastic optimisers. Of course, it is not immediately clear from the definition if there are any important differences between stochastic and deterministic optimisers. One key way in which stochastic optimisers differ from their deterministic counterparts is that their behaviour lacks an obvious on-policy/off-policy distinction. This is potentially an issue as we have previously focused on the on-policy behaviour of deterministic algorithms, showing that the on-policy behaviour implies a coherent prior over problems.

As a brief recap, we have seen that deterministic algorithms can be represented by a decision tree if we assume that they are always run starting with no data about the function being optimised. We call the behaviour described by this tree the algorithm's on-policy behaviour, as it contains all and only the situations the algorithm can encounter when following its own behaviour from a blank slate. Distinct from this on-policy behaviour is the so-called off-policy behaviour. The off-policy behaviour describes the behaviour of the algorithm in situations it would never encounter when following its own decisions. As an example consider an optimisation algorithm for functions $f : \{1,\dots,7\} \to \mathbb{N}$ that samples the x values in increasing order. First it would evaluate $f(1)$, then $f(2)$ and so on up to $f(7)$. Thus, after k function evaluations the algorithm would know exactly the k leftmost values of f. For example after three function evaluations the data might be $3,7,2,?,?,?,?$, and we would next sample $f(4)$, yielding say, $3,7,2,7,?,?,?$. An example of an off-policy situation for such an algorithm is $1,?,4,?,?,3,2$. This state of knowledge about f would never result from following the algorithm's own decisions starting from $?,?,?,?,?,?,?$. To summarise, for deterministic behaviour the on-policy decisions are the nodes of the tree starting from no data, the off-policy decision are all other decisions.

The problem with extending this idea to stochastic optimisers is that in the stochastic case there isn't necessarily a fixed decision tree starting from no data. Instead of data being either on-policy or off-policy as in the deterministic case, each particular data set now has some probability of being in the path the algorithm takes down the graph. In fact, we will now show that a stochastic optimiser implies a probability distribution over deterministic optimisers.

Theorem 27 (A stochastic optimiser implies a probability distribution over deterministic optimisers)**.** *A stochastic optimisation algorithm is equivalent in behaviour to a suitable distribution over deterministic optimisation algorithms. Specifically, running the stochastic optimiser on a problem is equivalent to sampling a deterministic optimiser from the distribution and then running that on the problem.*

Proof. A deterministic optimiser is defined by the decision it makes at each node. Any particular run of the stochastic optimiser will produce a set of decisions equivalent to some deterministic optimiser. More generally making a decision at every node produces a full deterministic behaviour (and thus necessarily also an on-policy behaviour). The probability of a full behaviour is just the product of the probabilities of the decisions at each node. The probability of an on-policy behaviour is the sum of the probabilities of the full behaviours that produce the on-policy behaviour of interest. □

An example of a run of a stochastic optimiser corresponding to a particular deterministic optimiser is given in Figure 4.12. We have seen that a stochastic algorithm is equivalent to a probability distribution over deterministic algorithms. Figure 4.13 shows the probability distribution over deterministic algorithms that is implied by the stochastic optimiser shown in Figure 4.11.

4.5.3 A Bayesian Characterising of Stochastic Optimisers

As we have seen, a stochastic optimisation algorithm S can be understood as a probability distribution Q over deterministic optimisation algorithms $A \in \mathcal{A}$. Each deterministic algorithm A can be understood as a probability distribution over functions, P_A, and so a stochastic algorithm can be thought of as a probability distribution over probability distributions over functions. This naturally produces a new probability distribution over functions, R, defined as:

$$R(f) = \sum_{A \in \mathcal{A}} Q(A) P_A(f) \tag{4.1}$$

As this R is itself a probability distribution over functions it corresponds to a particular deterministic algorithm, call the deterministic algorithm induced by a stochastic algorithm S in the above way A_S.

Thus, is in some sense the stochastic optimisation algorithm S is behaviour is informed by the probability distribution R. However, what algorithm actually does in a

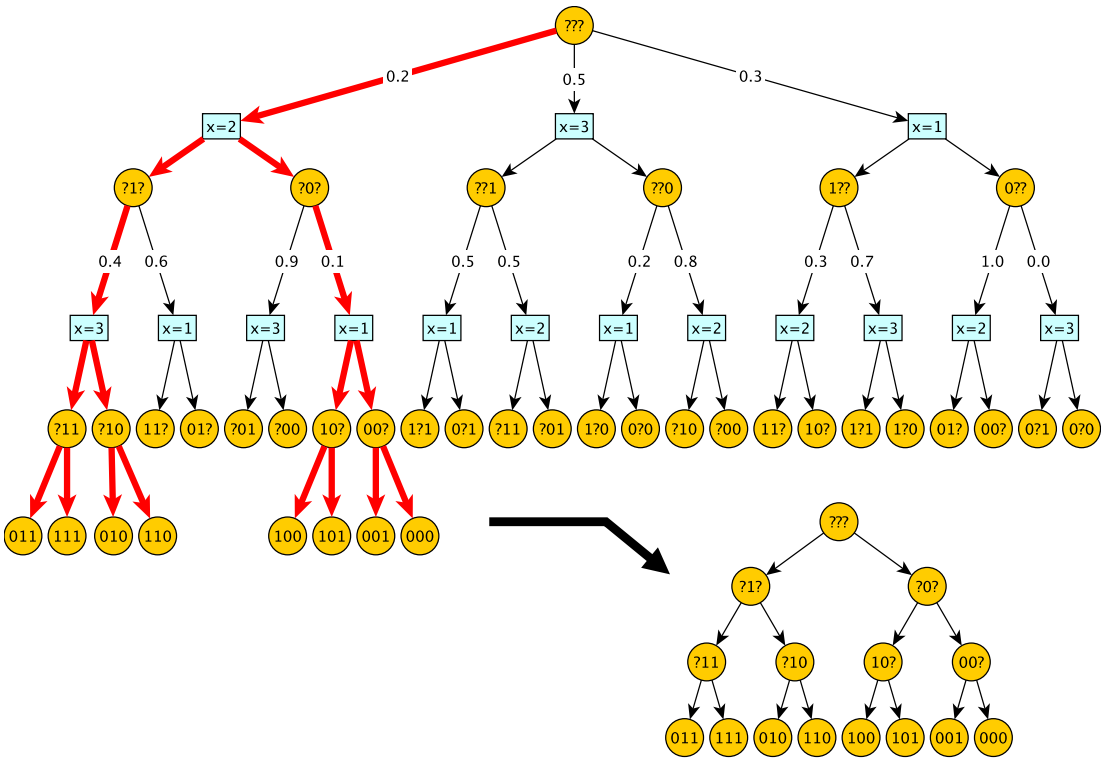

Figure 4.12: This diagram shows how a random decision at each node produces a deterministic algorithm behaviour tree. The red arrows in the upper tree show the choice made, and the tree on the bottom right is constituted from all and only these decisions. Choices only need to be made at nodes that might be encountered in the course of the optimisation if we are only interested in the resulting on-policy deterministic behaviour.

single run is, *behave optimally with respect to one of the many probability distribution that constitute* R. Every time the optimiser is run it randomly selects a constituent part of R with probability equal to the weight the constituent part in question is given in the construction of R.

The key idea that we have been perusing is that the behaviour of an algorithm reveals its beliefs about the sorts of problems it is likely to encounter. In particular, if we assume a given goal (in the form of a cost function) then the behaviour reveals a probability distribution over functions that the algorithm believes in, which we call the algorithm's prior. When we consider stochastic optimiser, it seems that behaviour is the result of a sample from a probability distribution over beliefs. This distribution over beliefs can be used to create a deterministic optimiser via Equation 4.1. We now examine whether any benefits can be had from the stochastic approach.

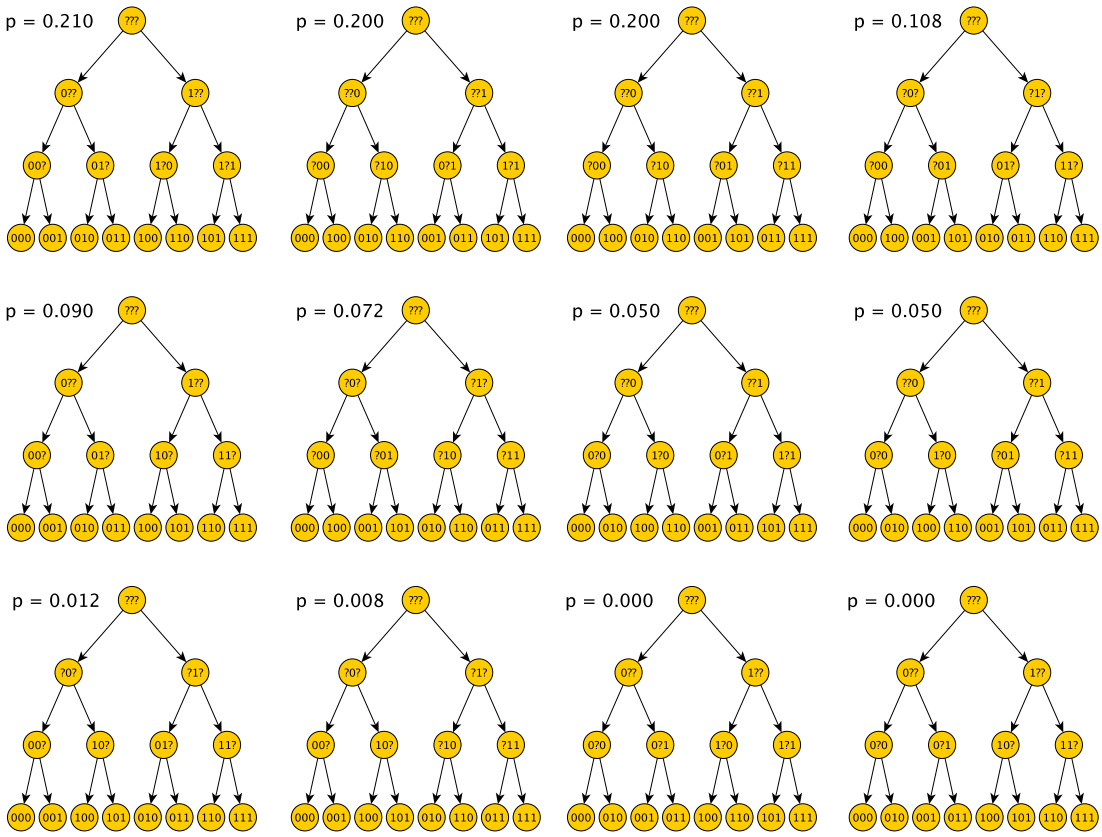

Figure 4.13: The stochastic optimiser in Figure 4.11 implies a probability distribution over deterministic optimisers. This diagram shows the decision tree and probability for each one of the twelve possible deterministic algorithms.

4.5.4 Are there Benefits to Stochastic Behaviour?

As we have noted above, stochastic optimisers are a strictly broader class of optimisation algorithms than deterministic optimisers. As a result, anything we could achieve with deterministic optimisers we can also achieve with stochastic optimisers. The question then is whether stochasticity enables us to achieve anything that is impossible with purely deterministic behaviour.

We have already seen that the no free lunch theorem holds for stochastic algorithms, and thus no stochastic algorithm is better than any other over all problems. In particular, as deterministic algorithms are a subset of stochastic algorithms this means that no stochastic algorithm is better than any deterministic algorithm over all problems. The no free lunch result then is that stochastic algorithms are no better behaviourally than deterministic ones. In this sense then, there are no benefits to stochastic algorithms. Further, as we saw in Chapter 2, various no free lunch extensions also

hold for stochastic optimisers, such as m-step no free lunch, sharpened no free lunch and focused no free lunch.

4.5.4.1 Non-Behavioural Benefits

That said however, the no free lunch algorithm is only concerned with the behaviour in a mathematical sense, i.e. the mapping that is performed. *In this sense we do not gain an advantage from allowing randomness*. Importantly though, no free lunch theorems do not address the practicality of the algorithm. Thus, although stochastic optimisation provides no behavioural benefits (and in fact has a disadvantage, as we will shortly see) computationally, and in terms of algorithm simplicity allowing randomness seems to be very advantageous, hence its wide application.

4.5.5 Are there Downsides to Stochastic Behaviour?

We have seen that there is no behavioural advantage from stochastic behaviour. In this section we show that actually random behaviour is in a sense less powerful that deterministic. This is a general result that is not ruled out by the no free lunch results seen in Chapter 2.

To see this let us first define the set of *strictly stochastic optimisers*; optimisers that are in the set of stochastic optimisers but are not in the set of deterministic optimisers. Strictly stochastic optimisers necessarily involve some randomness in their behaviour, whereas stochastic optimisers *can* involve randomness. Note that a strictly stochastic optimiser doesn't need to involve randomness in all of its decisions, just in at least one. We now state and prove a theorem that shows that for any strictly stochastic optimiser there is a deterministic optimiser that is expected to behave better the stochastic optimiser on any class of functions on which the stochastic optimiser performs well. The result comes about because, as we will see, stochastic optimisers mix Bayesian optimal belief updates with semi-random belief updates.

Theorem 28 (Strictly stochastic optimisers behave sub-optimally)**.** *A strictly stochastic optimiser mixes optimal Bayesian belief updating with random belief selection, resulting in behaviour that cannot be more effective than a deterministic optimiser.*

Proof. Assume that we are optimising problems that are in reality distributed according to some distribution P_r. A deterministic optimiser could in theory be constructed which behaved optimally with respect to the distribution P_r and a cost function of our

choosing. The optimiser would perform as well as possible, behaviourally, at the optimisation task. Consider now a strictly stochastic optimiser. It may well be that it sometimes produces the behaviour that is optimal given that the real distribution is P_r, but if it always did then it would not be behaving randomly, it would be the deterministic optimiser just discussed. Thus, its randomness must result potential deviation from the optimal strategy, by randomly behaving with respect to a sub-optimal (i.e. false) prior belief. □

4.6 Exploring Common Meta-Heuristic Approaches

In this section we direct our attention toward some common approaches to meta-heuristic optimisation with the aim of understanding how these approaches affect the underlying Bayesian structure. A good overview of meta-heuristic ideas can be found in [61]. Here we specifically consider hybrid optimisers and adaptive optimiser. Hybrid optimisers attempt to exploit the strengths of various other optimisers by combining them in some way, and adaptive optimisers are optimisation algorithms that change their behaviour over time base on the results of its exploration.

There is a dilemma that needs to be directly addressed when designing meta-heuristic optimisers. On the one hand, we know from the no free lunch theorems discussed in Chapter 2 that regardless of how we cleverly adjust, augment and combine optimisation algorithms we can never create a better algorithm in a totally general sense. Yet on the other hand, we know from experience that these clever algorithm adjustment strategies, often discussed in the literature under the umbrella of "meta-heursitic algorithms", do change the algorithms *in some way*. The algorithm isn't better in a fully general sense, but it is different. The dilemma then is this: How do we make new optimisers that are different in the right kind of way?

We argue that this is exactly the sort of problem the Bayesian interpretation is able to address. As we have shown over the previous chapters, all optimisers are representable as priors and cost functions. In this section we examine how meta-heuristic algorithm strategies change the underlying prior implied by the optimiser, and so adjust the type of problem with which the optimiser is best aligned.

We will see that meta-heuristic algorithms that intelligently switch between multiple optimisation algorithms or dynamically adapt parameters on-line are still optimisation algorithms, and thus themselves respect this rule. The prior which they use is composed in some way from the priors of the underlying algorithms. The exact nature

of this composition depends on the details of the meta algorithm. In the next two sections we discuss two popular meta mechanisms, parameter adaptation and algorithm gating.

4.6.1 Hybrid Optimisers

Hybrid optimisers attempt to exploit the strengths of various existing optimisation algorithms by combining them intelligently. We first consider the case where one of the optimisation algorithms is initially run with the express purpose of deciding which optimiser from a selection would be best for the problem at hand, and then the chosen algorithm is run as normal. We call this the algorithm selection meta-heuristic. We will see that this kind of meta-heuristic optimisation is a special type of prior combining, and, roughly speaking it allows sections of the prior corresponding to conditional scenarios to be defined by different optimisers.

4.6.1.1 Algorithm Selection

An algorithm selection meta-heuristic optimisation strategy initially runs an optimisation algorithm A_1 for a short time, it then uses those results to select an algorithm from algorithms $A_2, \ldots, A_n$ to run for the remainder of the optimisation.

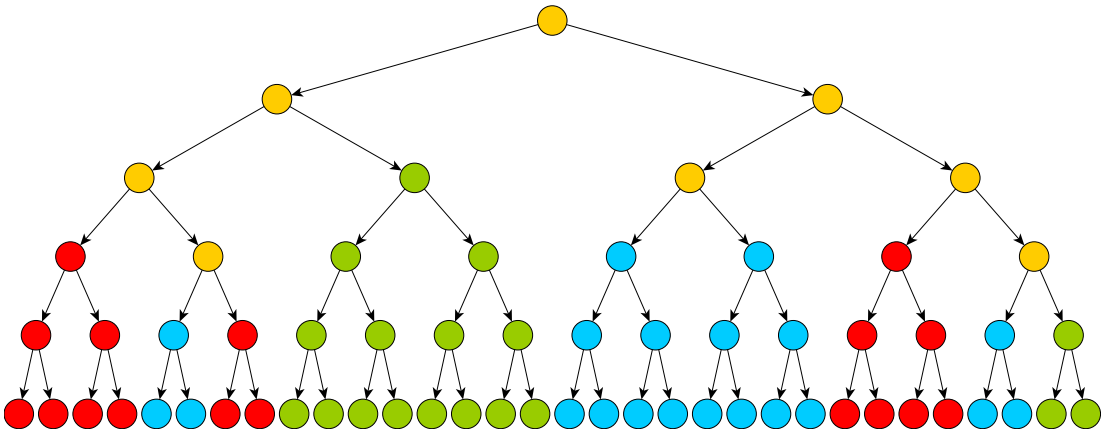

Figure 4.14: Algorithm selection. The initial algorithm (yellow) is run until it decides which of the thee algorithms (red, blue or green) is best suited to the problem, then it switches to that algorithm. The algorithm selection construction means that, in any given path down the tree the algorithm (ie, the colour) can change at most once.

A interesting question is, how is the resulting algorithm's prior related to the priors of the constituent algorithms? In Figure 4.14 we show how the behaviour tree of

the hybrid optimiser can be created by splicing together sections of the constituent optimiser's behaviour graphs.

Note that the sections of the behaviour graphs of $A_2, \ldots, A_n$ present in the resulting meta-optimiser's on-policy behaviour tree may not be from the on policy regions of the original optimisers. However, what can be seen by considering Figure 4.14 is that the resulting meta-heuristic optimiser's prior is constituted from conditional sections of $A_2, \ldots, A_n$, weighted so as to result in the behaviour of A_1 initially.

4.6.1.2 Algorithm Gating

An algorithm gating meta optimisation strategy uses a set of optimisation algorithms, A_1, A_2, …, A_n which it can switch between over the course of the optimisation. Of course, an algorithm created by switching between A_1, A_2, …, A_n is behaviourally identical to another non-gated optimisation algorithm, say B, regardless of how you implement the gating. What can we say about how the prior implied by B relates to the priors implied by A_1, A_2, …, A_n?

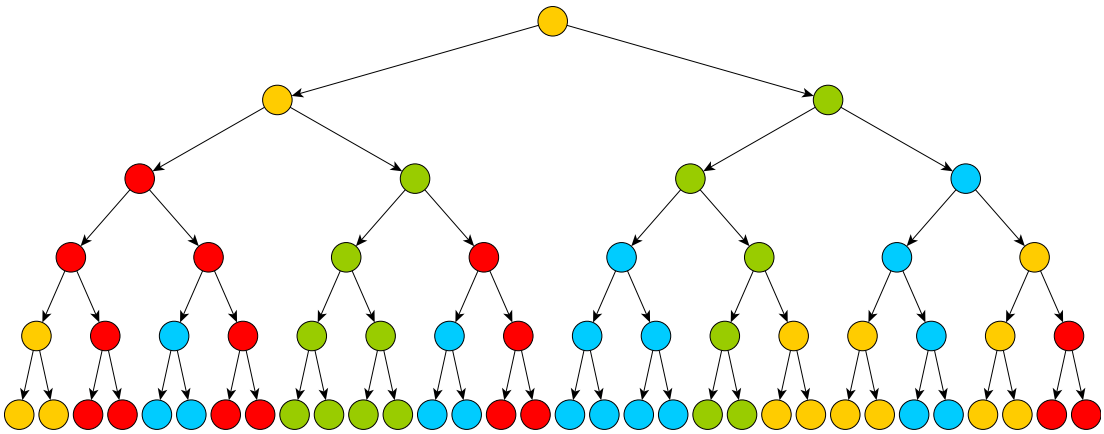

Figure 4.15: Optimisation algorithm B, shown with its composite parts in different colours. Note that in algorithm gating, in contrast to algorithm selection, a path down the tree can change algorithm (ie change colour) any number of times. For example the right most path down the tree goes from yellow to green to blue to yellow to red.

Again, as in algorithm selection, the sub-trees of the constituent algorithms used to construct the behaviour tree of B may well not come from their on-policy behaviour tree.

4.6.2 Adaptive Optimisers

There is an idea in optimisation that the algorithm may be able to adapt itself to the kind of problem at hand, and thus have a robust good performance over a broad range of problems.

We now turn to adaptive optimisers. Adaptive optimisers are optimisation algorithms that change their behaviour over the course of the optimisation based on the results of the exploration so far. Generally this is done though modulating some parameter(s) on which the algorithm's behaviour depends.

Parameter adapting meta-heuristic algorithms take a parametrised optimisation algorithm, A_θ, and augment it with a parameter adaptation rule, $z : D \to \theta$, which updates the algorithm's parameter vector θ on-line in response to the results of the samples seen so far. The intention is that the algorithm can adapt to the circumstances and exploit the information it gains about the objective function not just by selecting the next sample point intelligently, but also by changing the way it will select sample points in the future.

For any fixed parameter vector θ the algorithm's behaviour can be described by a prior over functions. Thus, the set of possible parameter vectors can be seen as describing a set of possible priors over functions. One view then is that the meta algorithm's prior changes as the parameter vector changes. This is a valid interpretation, but there is a less obvious alternative interpretation. For a deterministic parameter adaptation rule the meta algorithm has behaviour describable as a single prior over functions constituted from pieces of the priors from the aforementioned set. In this way we can see that parameter adaptation is an example of gating, which we discussed in Section 4.6.1.2.

4.6.2.1 Critical Optimisers

A critical optimiser is an algorithm that aims to optimally balances exploration and exploitation by maintaining a so-called critical state of activity. As an example we can consider a particle swarm type optimisation algorithm, which explores the problem space with a host of particles. Intuitively you want to avoid two situations: Firstly you do not want all the particles to collapse down into a single point, or a very small region of the parameter space. Secondly you do not want the particles to shoot off unrestricted in all directions.

Various critical optimisers have been proposed by researchers. Some of these are

critical variants of established meta-heuristics, for example adapting the standard particle swarm optimisation algorithm so that it self organises toward criticality [62]. Other optimisers are in some sense naturally critical. For example cuckoo search is naturally critical due to its hard-coded Levy flight jump dynamics, providing a random walk with power-law distributed step size [63].

Whether the criticality comes about though on-line adaptation or is a property of the optimiser generally, we can seek to understand how this critical balance between explosion and collapse is reflected in the algorithm's prior beliefs. This is a difficult question, and we will not be able to resolve it here. However, we make the following observation: It seems that the prior itself must maintain, in spite of being conditioned on the results as the optimisation progresses, a broadness that does not result on singling in on a focused X region, but at the same time does produce exploitation locally.

Chapter 5

Shattering

5.1 Introduction

Function optimisation research has two goals. The more obvious of the two is to create good algorithms for optimisation problems of various kinds. The other is to understand what makes optimisation algorithms "good", and how they should be compared and evaluated.

Here we consider an approach to the second challenge that draws inspiration from research on the Vapnik-Chervonenkis (VC) dimension and shattering [64], and is theoretically underpinned by the Bayesian framing of optimisers we have introduced in the previous chapters. A classification algorithm f parametrized by θ is said to shatter a set of points if for all possible labellings of the points as positive and negative there is a θ that results in f correctly classifying every point. The VC dimension of a classifier is the maximum number of points that can be arranged so that they are shatterable.

In this chapter we consider parametrized probability distribution over functions, and show that for these distributions we can find parameter settings which favour different optimisation algorithms. In particular, we show that any one of three popular optimisation algorithms can be made to out-perform the other two by selecting the sort of problem considered, and that by finding those problems on which an optimiser is better than its competitors, we learn about the sorts of problems it is suited to. This chapter can be seen as a case-study of no free lunch results which also has implications for the practical evaluation of optimisation algorithms.

As we covered in detail in chapter two, It has been known since the publication of Wolpert and Macready's no free lunch theorem in the late 90s that we can't have an optimisation algorithm that performs well on all possible problems [8], and instead the

best we can do is have an optimisation algorithm that is good at a subset of problems. Although this may seem disheartening for optimisation research, it is worth noting that this strong result holds only when the function to optimise is sampled froma block uniform distribution over the set of all function. In reality however, even a algorithm needing to handle a very broad range of problems is unlikely to be dealing with all problems with the required block probability. Instead, the hope is that there is common structure in the problems it encounters, and it is this structure that allows us to create optimisation algorithms that perform better than random guessing in general on real problems.

Moreover, for a given class of real-world problems some algorithms *are* better than others. In these cases their is a good match between structure the algorithm can exploit and the structure actually present in the problems. Ideally when selecting algorithms for a problem we would try to optimise this matching of algorithm strength and problem type. In order to do this, and thus make the best use of available algorithms, it is necessary to understand the kinds of problems each algorithm is particularly suited to. In this chapter we present an approach to this problem, more specifically, we try to address the following question: Given an optimisation algorithm, how can we learn the sorts of problems it excels at?

Another question is pertinent here. What sorts of problems do we want optimisation algorithms to be good at? The natural answer of course is that we want algorithms to be good at the sort of problems on which they are likely to be used. Although this answer is no doubt true, it is not very useful. Thus, we suggest the following more concrete approximation to the above intuitive response: each algorithm was designed to be good in some sense, to solve a particular type of problem that someone needed to solve for example. Thus, if we knew the sort of problems algorithms were collectively good at, we might start to get an understanding of the problems that arise in reality. Rather than making a algorithm good at everything, which we know to be impossible, the aim could be to make an algorithm that is better than other algorithms at what they are good at. Assuming that what they are good at is useful, then this new algorithm should be a useful tool. The cost of course, as we know from no free lunch theorem, is that the new algorithm will be even worse at the things the original algorithms were bad at.

There has been previous related work, in which researchers tried to evolve via a genetic algorithm problem functions that a particular optimiser found difficult to optimise [65, 66, 67]. This was doen to find weaknesses in optimisers. Here however

we are interested in when an algorithm does well. All optimisers have weakness, and we focus on understanding their strengths.

We now formally introduce a concept of shattering for optimisation algorithms. Let $\mathcal{A}$ be a set of optimisation algorithms and let $\mathcal{P}$ be a set of probability distributions over functions. We say $\mathcal{P}$ shatters $\mathcal{A}$ iff $\forall A \in \mathcal{A}, \exists P \in \mathcal{P}\ s.t.\ \forall \hat{A} \neq A \in \mathcal{A}, E[\hat{A}|P] > E[A|P]$. Where $E[A|P]$ is the expected performance of A on functions sampled from P, under some performance metric. In other words, a set of probability distributions shatters a set of optimisation algorithms iff each optimisation algorithm is better than the others on average on at least one probability distribution.

5.2 Proof of Concept

Above we defined shattering in abstract, we now give a concrete example. In our example the problems (or test functions) are sampled from the following parametrized probability distribution over possible functions. Parameters $a,b,c \in \mathbb{R}^+$ are set, then a function $f : \mathbb{R}^n \to \mathbb{R}$ is created by uniformly sampling 200 points, $p_1,\ldots,p_{200}$ from $[-50,50]^n \subset \mathbb{R}^n$ and 5 points, $r_1,\ldots,r_5$ from $[0,1]^n \subset \mathbb{R}^n$. With these values fixed, f is defined as follows:

$$f_{a,b,c}(x) = k_{a,b}(x) + h_c(x)$$

$$h_c(x) = \frac{1}{4}\sum_{i=1}^{200} \sin(c|p_i - x|r_{1_i}2\pi + r_{2_i}2\pi)$$

$$k_{a,b}(x) = \frac{-2}{1+e^{-\lambda_{a,b}(x)}} + 1$$

$$\lambda_{a,b}(x) = b\sum_{i=1}^{200} r_{3_i}(a+r_{4_i})\sin(2\pi(r_{5_i} + \frac{|p_i - x|}{a+r_{4_i}}))$$

Thus, the probability distribution is parametrized by $\{a,b,c\}$ and we sample from it by drawing $p_1,\ldots,p_{200}$ and $r_1,\ldots,r_5$ from uniform distributions as described above. We write $f = P_f(\alpha,\beta,\gamma)$ to mean f is sampled from the distribution described above with the given parameters, i.e. $a = \alpha, b = \beta, c = \gamma$.

This parametrised probability distribution is not the only possibility. It was designed to allow different parameter choices to create very different kinds of functions. It necessarily only covers a small part of function space, but as we show below, this

is enough to achieve our goal. Broadly speaking the h term provides larger slower variations to which the k term adds clipped noise of various strengths and frequencies. Figures 5.1 and 5.2 show examples of functions k and h respectively for various parameter choices. Figure 5.3 show a selection of example 1D and 2D functions sampled from the final distribution.

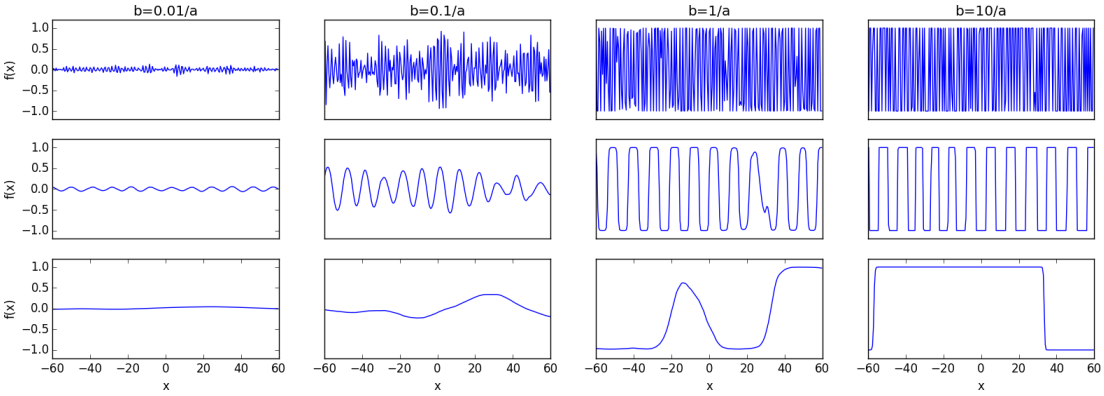

Figure 5.1: $k(a, b)$ for various values of a and b. Note that the values of b are functions of the values of a.

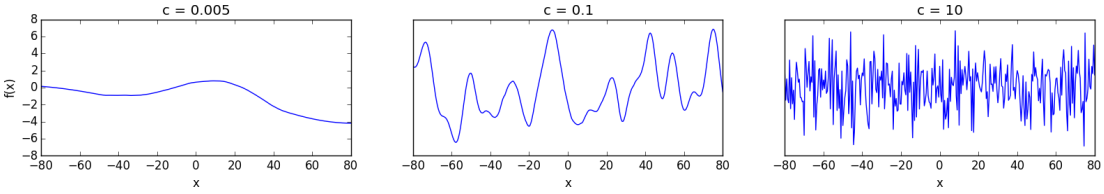

Figure 5.2: $h(c)$ for three values of c. Note that unlike k, the output of h is not always between -1 and 1.

This is the space of function distributions we will use to compare the optimisation algorithms.

5.2.1 Experimental Details

We used 10 dimensional functions in all experiments, the algorithms were allowed 5000 function evaluations in total. All function evaluations counted towards this limit, including the samples taken for numerical approximation of the gradient. For each parameter set ten functions were generated, then for each of these functions the mean and variance over ten runs was taken.

5.3 Algorithms

We have chosen three algorithms to compare using our method, these are the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS), simulated annealing (SA) and particle swarm optimisation (PSO). These three algorithms were chosen as they each work in a distinct way. BFGS is a powerful gradient descent algorithm, SA is an effective probabilistic technique, and PSO uses the emergent behaviour of a swarm of agents. We now briefly introduce each of the algorithms.

5.3.1 Broyden-Fletcher-Goldfarb-Shanno

BFGS is a Quasi-Newton method which searches for optima using numerical gradient approximations and line searches. We use a python implementation from the SciPy library [68], which follows the description in "Wright, and Nocedal 'Numerical Optimization', 1999" [69]. Essentially it is a hill-climbing method, that is known to produce good results fairly reliably.

5.3.2 Simulated Annealing

Simulated annealing takes its name from chemical annealing techniques. The function being optimised is explored by making random "jumps" from the current solution and (stochastically) updating the optimum if better positions are found. The method was first introduced in 1983 by Kirkpatrick et. al. [70] and is still popular. A temperature parameter scales the jump size, and is decreased to zero over the course of the optimisation. The result of this decreasing temperature is a tendency for the search to become more local over time. We use the following simple simulated annealing algorithm, where T is the maximum number of time-steps, t is the current time-step, b is the current best position and $N(\mu,\sigma^2)$ is a sample from a normal distribution with mean μ and standard deviation σ.

$$p_t = N(b,(T-t)c/T)$$

$$\text{if } f(p_t) < f(b) \text{ then } b = p_t$$

5.3.3 Particle Swarm Optimization

PSO is a popular meta-heuristic optimisation technique. Originally inspired by the flocking behaviour of birds, the algorithm simulates a swarm of agents exploring the function space [71]. Each agent's exploration is based both on the global best, and the agent's personal best, with parameters determining the relative importance of these two influences. Here we use a heterogeneous PSO where these parameters are randomly selected for each particle. Specifically, each particle has two parameters α_1 and α_2, α_1 is set uniform randomly in $[0,4]$, and α_2 is set to be $4-\alpha_1$. In addition, each particle has a starting inertia, ω_0 uniform randomly sampled from $[0.4,0.8]$. At time-step t $\omega_t = \omega_0 - 0.4 \times \frac{t}{T}$ where T is the total number of steps the optimiser will be run for. These are standard parameter ranges from the literature. We use the heterogeneous PSO variant here rather than the standard PSO as it tends to more quickly find reasonable results (sometimes paying for this by doing less well in the long run), and this is suitable here as we only run the optimisers for 5000 samples in our experiment. Let v_t, p_t be the particle's velocity and position respectively at time t and let b, g be the particles personal best position found and the best position found among all particle respectively. Then we update the particle as follows:

$$v_{t+1} = v_t\omega_t + \alpha_1 u_1(b-p_t) + \alpha_2 u_2(g-p_t)$$

$$p_{t+1} = p_t + v_{t+1}$$

Where u_1, u_2 are independent uniform random sample from $[0,1]$.

5.3.4 Algorithm Differences

The algorithms introduced above consist of a modern hill climbing optimiser, a stochastic optimiser and a swarm approach. Broadly speaking, the BFGS algorithm attempts to move efficiently to the optimum by exploiting gradient information, SA is a random search that focuses in around promising results over the course of the optimisation, and PSO preferentially explores the regions between and around pairs of strong candidate solutions. Each of these approaches has its own particularities, and here we are interested in how these will shape the resulting prior distributions.

5.4 Results

The performance of three optimisation algorithms was compared on various points in the space of probability distributions over functions. First we searched for parameters that favoured each of the three algorithms. Such parameter sets were found, and the results are shown in Figure 5.4. Finding such points serves as a constructive proof that our set of three optimisation algorithms is shattered by our set of priors over functions.

By examining the figures produced from the probability distributions found (Figure 5.4) we can comment on the relative strengths of the algorithms. For the BFGS optimiser, the best suited functions seem to have medium length scale smooth variations with little global structure. The PSO dominates when slow changing global structure is riddled with small scale sharp fluctuations between extremes. Finally, SA is best on large scale fluctuations that are partially noised at the small scale.

5.4.1 A Meta-Algorithm

When presented with the results above, showing that each algorithm excels on particular problems, a natural question is whether this knowledge can be exploited to produce a meta-algorithm that intelligently selects the best algorithm for the problem. Here we use a simple method to produce such a meta-algorithm, and show that it does indeed outperform the three original algorithms on average.

For the meta-algorithm we split the optimisation into two stages, the first 500 function evaluations are used to determine which optimiser is best to use, then the selected optimiser is run for the remaining 4500 function evaluations. As a simple selection technique, we make use of the fact that BFGS has very different expected results after 500 evaluations on each of the three parameter sets. Thus, we run BFGS for 500 evaluations, then use the result obtained to decide which optimiser to run for the remainder of the optimisation. The result of this meta optimiser and also the individual algorithm is shown in Figure 5.5.

5.5 Conclusion

We have shown that any one of three popular optimisation algorithms can be made to out-perform the other two by selecting the sort of problem considered. This sort of knowledge is important when choosing optimisation algorithms for a particular task. We have begun to uncover the prior beliefs of the meta-heuristic optimisers.

An interesting follow up investigation would be to test a set of function optimisation algorithms on functions sampled from a Gaussian-process for various choices of parameters and kernels. This would provide further insight into the sorts of function the various algorithms excel in optimizing.

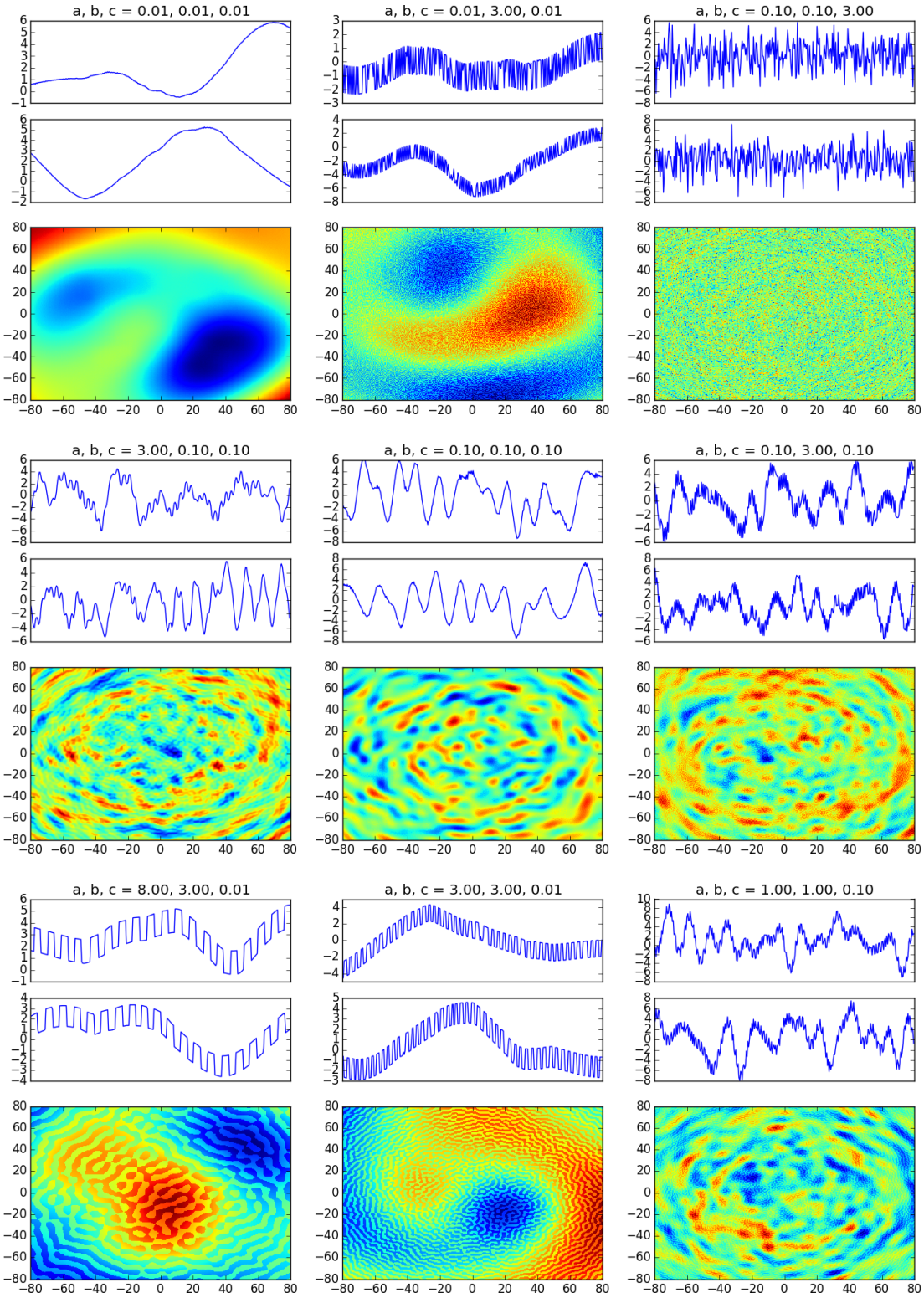

Figure 5.3: One and two dimensional example functions for nine different parameter choices. As the examples show, a wide variety of functions can be produced.

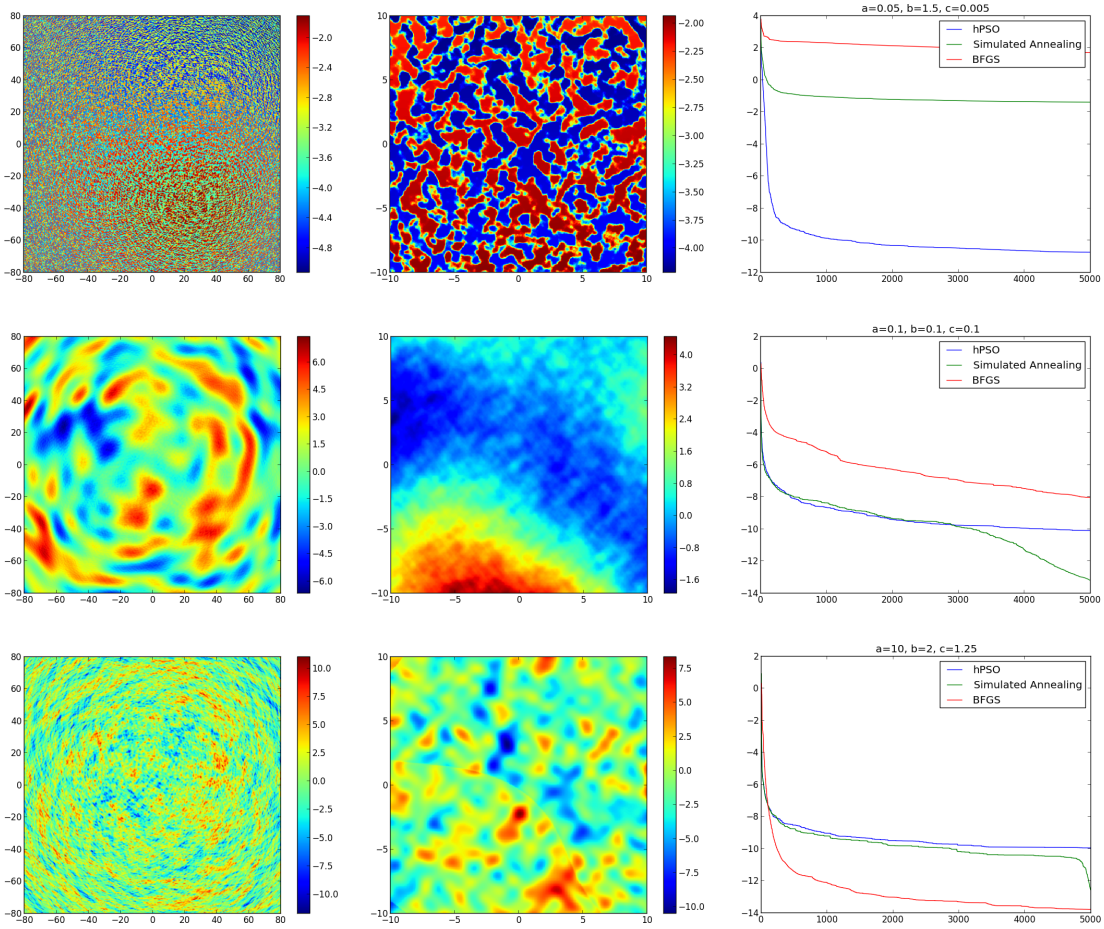

Figure 5.4: The first row shows an example function sampled from the distribution with parameters tuned such that hPSO out-performs the other optimisation algorithms. The second row shows a function where simulates annealing outperforms, and the last rows shows a function for which the BFGS optimisation performs best. Each row shows a 2D cross-section of a 10D function, a zoomed view of the same function, and then the algorithm's performance on functions sampled from the distribution with parameters given, averaged over ten sample functions each optimisation ten times.

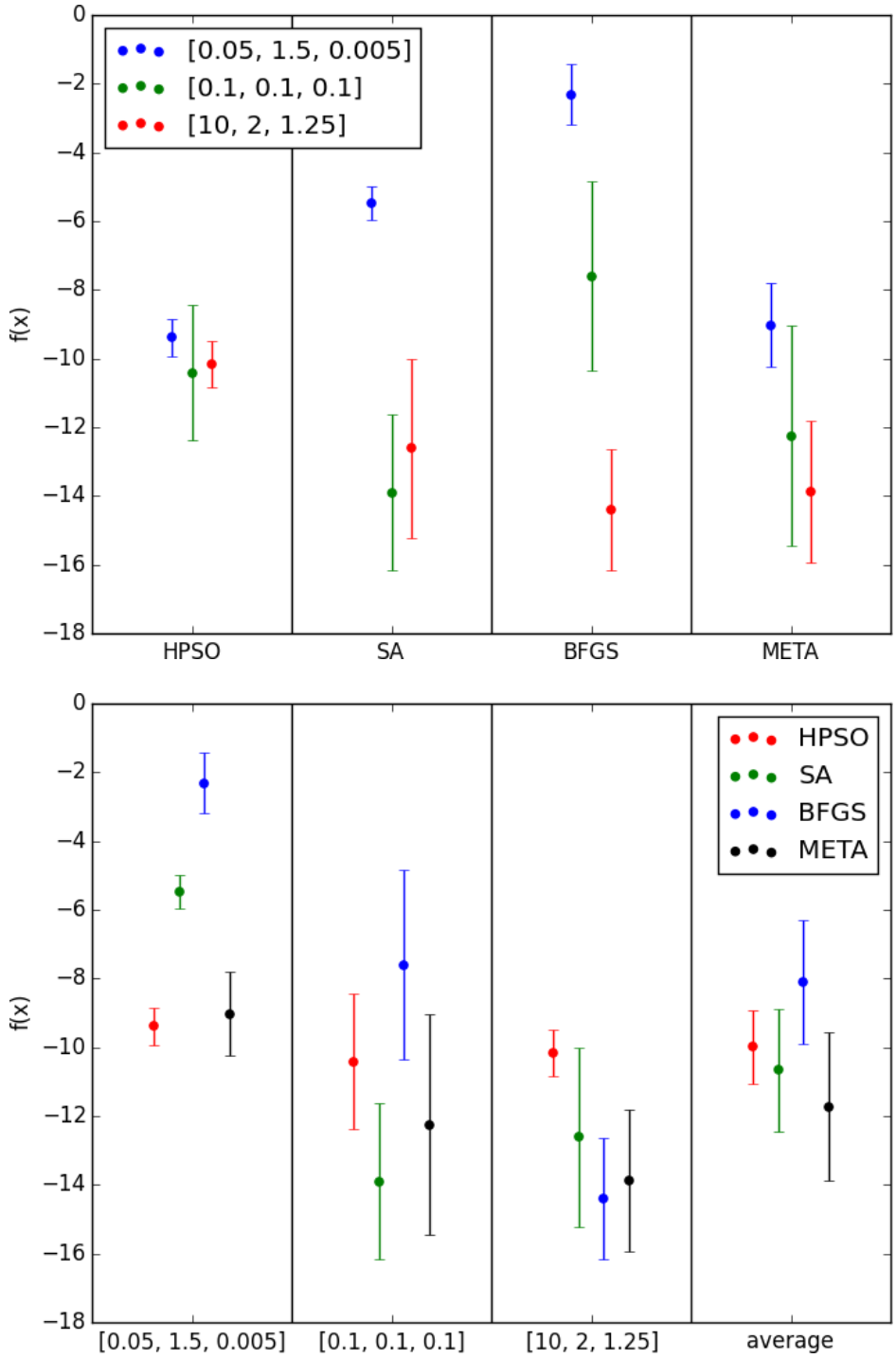

Figure 5.5: Side-by-side comparison of the final results obtained by the three optimisation algorithms and the meta optimiser on the three parameter settings. For each of the parameter settings ten functions were sampled and each optimisation algorithm ran ten times on each function. Mean values and error bars are plotted. Observe that the meta optimiser is competitive on all three problem types.

Chapter 6

Conclusion

6.1 Further Applications

We have already examined some uses of the Bayesian framework in the latter half of Chapter 4, and in Chapter 5. In this section we explore other potential applications. As we will see, there are various possible directions that further investigation could take.

6.1.1 Meta-heuristics as Bayesian Optimisers

One key result of this work is that it opens the door to the use of meta-heuristic optimisers as fast but theoretically well understood Bayesian optimisers. As we have seen, all optimisers are Bayesian, it is just that in most cases the optimiser's prior is compressed into its program in a highly obfuscated way. Now that it is clear that it always makes sense to talk about an optimisers prior, it opens up the possibility of trying to accurately determine the prior for a given optimiser. For example, it would be both interesting and potentially very useful to have a better understanding of the implied beliefs resulting from particle swam optimisers with different parameter choices. This approach could potentially formalise the still somewhat "dark art" that is meta-heuristic parameter tuning.

6.1.2 Approximate Bayesian Inference

Another possibility is that fast meta-heuristic optimisers could be tuned, by parameter adjustment, to maximally align with slower Gaussian process optimisers, and then used as a high speed approximate alternative. Moreover, if we understood the priors of

meta-heuristic optimisers well enough we would know in what way our approximate results were biased.

6.1.3 Learning and Evolving Optimisers

As we briefly touched on in Section 4.2, further investigation into algorithm behaviour regions could lead to interesting results in connectivity structure. For example, suppose you have an optimiser behaving with respect to a particular prior, how would small variations in that prior affect the optimisers behaviour? Are there generally many behaviours that are next to a given behaviour, or are behaviours often somewhat isolated except for a few neighbours? These sorts of questions are directly relevant to research into evolution of optimisation algorithms, and other approaches to learning to optimise.

6.1.4 The Bayesian Brain

Investigation into how behaviour relates to underlying beliefs is also at the centre of the Bayesian brain hypothesis, which is broadly the idea that brains are performing Bayesian inference [72, 73]. In this field of study it is again a case of trying to infer underlying beliefs from the examination of behaviour. We have seen here that every on-policy behaviour is Bayesian, and so the brain, when performing an optimisation task, is in some sense necessarily Bayesian, although what "on-policy" might mean in terms of brains is not an easy question.

6.1.5 Blending and Interpolating Meta-Heuristics

Another rich potential application is the developing of an "arithmetic" for optimisers. The idea is that now all optimisers can be understood as points in the same space (prior distribution with respect to an admissible cost function), concepts like the average optimiser, the difference between optimisers, and the sum of optimisers can all be made rigorous. Further, vector space ideas can be applied. Suppose optimiser A is better at a task than optimiser B, then we could meaningfully ask about optimiser $A + (A - B)$. Similarly, the approach provides a way to meaningfully ask "is optimiser X more similar to optimiser Y or optimiser Z", which could then be useful for choosing which algorithms to try on a problem.

As well as the potential to develop a better understanding of how algorithms can

be interpolated between in the ways discussed above, we can also investigate a more direct mixing of optimisers. For example, if we examined the result of running two optimisers in step, such that they each take turns sampling.

Thus, in these and other ways there is potential to understand not just particular algorithms, but rather the space of behaviours which a collection of algorithms span.

6.1.6 Automatically Developing Specialised Optimiser

Given a specific problem class, defined by a distribution over problems and a cost function, we have seen that there is, in theory at least, an optimal optimiser. However, there is currently no general way to automatically construct an optimiser specialised for a particular problem. Even if we were able to construct the optimal behaviour optimiser, it might be the case that the complexity of the problem distribution necessarily resulted in a very slow running algorithm.

Given sufficient understanding of the priors of existing meta-heuristics, and knowledge of how these algorithms could be combined and extrapolated, as discussed in Section 6.1.5, an automated algorithm constructor could be in theory be produced, which would take as an input a very specific problem class, and which would produce a fast approximately Bayesian optimal optimiser for the input problem class by tuning and combining various well understood meta-heuristics.

6.2 Summary

In this thesis we have looked at optimisation algorithms, in particular, we have investigated how their behaviour is best understood. We saw that no free lunch shows universal optimisers to be impossible, and instead the alignment between problem and optimiser is all that matters behaviourally. We made this concept of alignment rigorous through a Bayesian understanding of behaviour. However, although the Bayesian approach to optimisation is known to be theoretically powerful, it is generally thought to apply only to a sub-set of meta-heuristic optimisers, namely those that explicitly behave in a Bayesian way. In fact, we have shown that all deterministic, forgetful, and stochastic optimisers are necessarily Bayesian, that is, they behave Bayesian optimally with respect to a prior and cost function pair.

Specifically, in this thesis we have provided various extensions and refinements to the no free lunch theorems, including extension to algorithms with arbitrary stopping

conditions 2.5.2, off-policy behaviour 2.6.6, restricted metrics 2.6.7, parallelism 2.7.7, minimax behaviour 2.8.7 and order statistics 2.9.9. We have show that all deterministic on-policy behaviour is Bayesian 3.5, and moreover, that different cost functions produce different prior distributions 3.8. Further, we have show that some cost functions can be used to produce any on-policy behaviour 3.8, and that the Kolmogorov complexity of the implied prior is related to the Kolmogorov complexity of the optimisation algorithm 3.7.

We then further examined this Bayesian approach, showing that uniform randomly selecting a prior is not the same as uniform randomly selecting a behaviour 4.2.2.1. We considered in detail the issues related to full behaviours 4.3.2. We then extended our framework to forgetful optimisers in 4.4 and stochastic optimisers in 4.5.5, showing in both cases that the behaviour can be seen as Bayesian. We also saw that stochastic optimisers can be worse behaviourally than deterministic optimisers, and can never be behaviourally better 4.5.5.

We then used the Bayesian framework to examine various approaches to meta-heuristic optimiser construction, such as algorithm gating, and on-line parameter adaptation 4.6.1, and lastly we introduced the idea of algorithm shattering and gave a proof of concept example 5.2, which also served as a case studied of the real life importance of no free lunch results.

The fact that this Bayesian framing is universally applicable gives the various diverse approaches to optimisation a unified underlying framework. It also gives us a clear inroad to a better understand the strengths of existing optimisers. Moreover, it shows that when developing new meta-heuristics, behaviourally, the single defining feature is the implied prior.

List of Figures

Bibliography

[1] Yu-Chi Ho and David L Pepyne. Simple explanation of the no-free-lunch theorem and its implications. *Journal of Optimization Theory and Applications*, 115(3):549–570, 2002.

[2] Darrell Whitley. Sharpened and focused no free lunch and complexity theory. In *Search Methodologies*, pages 451–476. Springer, 2014.

[3] Xiaodong Li, Ke Tang, Mohammad N Omidvar, Zhenyu Yang, and Kai Qin. Benchmark functions for the cec 2013 special session and competition on large-scale global optimization. *gene*, 7(33):8, 2013.

[4] Daniel Bratton and James Kennedy. Defining a standard for particle swarm optimization. In *Swarm Intelligence Symposium*, pages 120–127. IEEE, 2007.

[5] Emile Contal, Vianney Perchet, and Nicolas Vayatis. Gaussian process optimization with mutual information. *arXiv preprint arXiv:1311.4825*, 2013.

[6] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[7] David H Wolpert and William G Macready. No free lunch theorems for search. Technical report, SFI-TR-95-02-010, Santa Fe Institute, 1995.

[8] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[9] Cullen Schaffer. A conservation law for generalization performance. In *Proceedings of the 11th International Conference on Machine Learning*, pages 259–265, 1994.

[10] David H Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

[11] David H Wolpert. On the connection between in-sample testing and generalization error. *Complex Systems*, 6(1):47, 1992.

[12] David H Wolpert. On overfitting avoidance as bias. Technical report, SFI TR 92-03-5001. Santa Fe, NM: The Santa Fe Institute, 1993.

[13] Darrell Whitley and Jonathan Rowe. Focused no free lunch theorems. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pages 811–818. ACM, 2008.

[14] Darrell Whitley. Functions as permutations: regarding no free lunch, walsh analysis and summary statistics. In *Parallel Problem Solving from Nature PPSN VI*, pages 169–178. Springer, 2000.

[15] Joseph C Culberson. On the futility of blind search: An algorithmic view of "no free lunch". *Evolutionary Computation*, 6(2):109–127, 1998.

[16] Tor Lattimore and Marcus Hutter. No free lunch versus occam's razor in supervised learning. In *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence*, pages 223–235. Springer, 2013.

[17] Loris Serafino. No free lunch theorem and bayesian probability theory: two sides of the same coin. some implications for black-box optimization and metaheuristics. *arXiv preprint arXiv:1311.6041*, 2013.

[18] Thomas English. No more lunch: Analysis of sequential search. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 227–234. IEEE, 2004.

[19] Thomas M English. Optimization is easy and learning is hard in the typical function. In *Evolutionary Computation. Proceedings of the 2000 Congress on*, volume 2, pages 924–931. IEEE.

[20] Thomas English. On the structure of sequential search: beyond "no free lunch". In *Evolutionary Computation in Combinatorial Optimization*, pages 95–103. Springer, 2004.

[21] Edgar A Duéñez-Guzmán and Michael D Vose. No free lunch and benchmarks. *Evolutionary Computation*, 21(2):293–312, 2013.

[22] Nicholas J Radcliffe and Patrick D Surry. Fundamental limitations on search algorithms: Evolutionary computing in perspective. In *Computer Science Today*, pages 275–291. Springer, 1995.

[23] C Schumacher, Michael D Vose, and L Darrell Whitley. The no free lunch and problem description length. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 565–570, 2001.

[24] Christian Igel and Marc Toussaint. A no-free-lunch theorem for non-uniform distributions of target functions. *Journal of Mathematical Modelling and Algorithms*, 3(4):313–322, 2005.

[25] Stefan Droste, Thomas Jansen, and Ingo Wegener. Optimization with randomized search heuristics – the (a) nfl theorem, realistic scenarios, and difficult functions. *Theoretical Computer Science*, 287(1):131–144, 2002.

[26] Evan J Griffiths and Pekka Orponen. Optimization, block designs and no free lunch theorems. *Information Processing Letters*, 94(2):55–61, 2005.

[27] David W Corne and Joshua D Knowles. No free lunch and free leftovers theorems for multiobjective optimisation problems. In *Evolutionary Multi-Criterion Optimization*, pages 327–341. Springer, 2003.

[28] Travis C Service. A no free lunch theorem for multi-objective optimization. *Information Processing Letters*, 110(21):917–923, 2010.

[29] David Corne and Joshua Knowles. Some multiobjective optimizers are better than others. In *Evolutionary Computation. The 2003 Congress on*, volume 4, pages 2506–2512. IEEE.

[30] Tom Everitt. Universal indution and optimisation: No free lunch? 2013.

[31] David H Wolpert. What the no free lunch theorems really mean; how to improve search algorithms. In *Santa fe Institute Working Paper*, page 12. 2012.

[32] Matthew J Streeter. Two broad classes of functions for which a no free lunch result does not hold. In *Genetic and Evolutionary Computation-GECCO*, pages 1418–1430. Springer, 2003.

[33] Christian Igel and Marc Toussaint. On classes of functions for which no free lunch results hold. *arXiv preprint cs/0108011*, 2001.

[34] Steffen Christensen and Franz Oppacher. What can we learn from no free lunch? a first attempt to characterize the concept of a searchable function. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1219–1226, 2001.

[35] David H Wolpert and William G Macready. Coevolutionary free lunches. *Evolutionary Computation, IEEE Transactions on*, 9(6):721–735, 2005.

[36] Stefan Droste, Thomas Jansen, and Ingo Wegener. Perhaps not a free lunch but at least a free appetizer. Technical report, Universität Dortmund, 1998.

[37] Tom Everitt, Tor Lattimore, and Marcus Hutter. Free lunch for optimisation under the universal distribution. In *Evolutionary Computation (CEC), Congress on*, pages 167–174. IEEE, 2014.

[38] Darrell Whitley and Jean Paul Watson. Complexity theory and the no free lunch theorem. In *Search Methodologies*, pages 317–339. Springer, 2005.

[39] Yossi Borenstein and Riccardo Poli. Kolmogorov complexity, optimization and hardness. In *Evolutionary Computation. Congress on*, pages 112–119. IEEE, 2006.

[40] Shai Ben-David, N Srebro, and R Urner. Universal learning vs. no free lunch results. In *Philosophy and Machine Learning Workshop NIPS*, 2011.

[41] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

[42] James L Carroll. No free-lunch and bayesian optimality. In *IJCNN Workshop on Meta-Learning*. Citeseer, 2007.

[43] Loris Serafino. Optimizing without derivatives: What does the no free lunch theorem actually say? *Notices of the AMS*, 61(7), 2014.

[44] Klaus Mosegaard. Limits to nonlinear inversion. In *Applied Parallel and Scientific Computing*, pages 11–21. Springer, 2012.

[45] Anne Auger and Olivier Teytaud. Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, 57(1):121–146, 2010.

[46] J Rowe, M Vose, and A Wright. Reinterpreting no free lunch. *Evolutionary Computation*, 17(1):117–129, 2009.

[47] Aureli Alabert, Alessandro Berti, Ricard Caballero, and Marco Ferrante. No-free-lunch theorems in the continuum. *arXiv preprint arXiv:1409.2175*, 2014.

[48] David H Wolpert. The supervised learning no-free-lunch theorems. In *Soft Computing and Industry*, pages 25–42. Springer, 2002.

[49] Huaiyu Zhu and Richard Rohwer. No free lunch for cross-validation. *Neural Computation*, 8(7):1421–1426, 1996.

[50] Yu-Chi Ho. The no free lunch theorem and the human-machine interface. *Control Systems, IEEE*, 19(3):8–10, 1999.

[51] Xin-She Yang. Free lunch or no free lunch: that is not just a question? *International Journal on Artificial Intelligence Tools*, 21(03):1240010, 2012.

[52] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, 21(4):345–383, 2001.

[53] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[54] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[55] Carl Edward Rasmussen and Chris Williams. Gaussian processes for machine learning. 2006.

[56] Ming Li and Paul MB Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer, 2009.

[57] James Robert Lloyd, David Duvenaud, Roger Grosse, Joshua B Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models. *arXiv preprint arXiv:1402.4304*, 2014.

[58] Bailu Si, J Michael Herrmann, and Klaus Pawelzik. Gain-based exploration: from multi-armed bandits to partially observable environments. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 1, pages 177–182. IEEE, 2007.

[59] Edward Snelson and Zoubin Ghahramani. Local and global sparse gaussian process approximations. In *International Conference on Artificial Intelligence and Statistics*, pages 524–531, 2007.

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

[61] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[62] Morten Lovbjerg and Thiemo Krink. Extending particle swarm optimisers with self-organized criticality. In *wcci*, pages 1588–1593. IEEE, 2002.

[63] Xin-She Yang and Suash Deb. Cuckoo search via lévy flights. In *Nature & Biologically Inspired Computing. World Congress on*, pages 210–214. IEEE, 2009.

[64] Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[65] William B Langdon and Riccardo Poli. Evolving problems to learn about particle swarm optimizers and other search algorithms. *Evolutionary Computation, IEEE Transactions on*, 11(5):561–578, 2007.

[66] William B Langdon and Riccardo Poli. Evolving problems to learn about particle swarm and other optimisers. In *Evolutionary Computation. The Congress on*, volume 1, pages 81–88. IEEE, 2005.

[67] William B Langdon, R Poll, Owen Holland, and Thiemo Krink. Understanding particle swarm optimisation by evolving problem landscapes. In *Swarm Intelligence Symposium. Proceedings of the*, pages 30–37. IEEE, 2005.

[68] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2014-12-23].

[69] SJ Wright and J Nocedal. *Numerical optimization*, volume 2. Springer New York, 1999.

[70] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[71] James Kennedy, Russell Eberhart, et al. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948. Perth, Australia, 1995.

[72] David C Knill and Alexandre Pouget. The bayesian brain: the role of uncertainty in neural coding and computation. *TRENDS in Neurosciences*, 27(12):712–719, 2004.

[73] Kenji Doya. *Bayesian brain: Probabilistic approaches to neural coding*. MIT press, 2007.